

APACHE, PHP-FPM & NGINX

How to Build a Secure, Fast and
Powerful Web-Server!



APACHE



Adrian Ling

www.ilovevirtualmin.com

Apache, PHP-FPM & Nginx Reverse-Proxy

How to Build a Secure, Fast &
Powerful Webserver with
emphasis on Web Security.

(Practical Guide Series Book #3)

Adrian Ling Kong Heng

Copyright © 2015

Copyright 2015 © Adrian Ling Kong Heng. All rights reserved.

No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage or retrieval system without prior written permission from the copyright owner and the publisher.

Acknowledgements

I am grateful to God who enabled me to write this book, and to whom I owe my very existence.

Special thanks to my beautiful wife and my 2 wonderful kids for supporting me and giving me the time and space to write.

And of course, this book would not have been possible without the great folks who created the Apache Web-Server, the PHP Programming Language (and PHP-FPM) and the Nginx Web-server. Hats off to the countless other programmers and developers who continually work on these great software to make them even better, faster and more secure with each passing day.

Disclaimer and Terms of Use Agreement

The author and publisher of this eBook and the accompanying materials have used their best efforts in preparing this eBook. The author and publisher makes no representation or warranties with respect to the accuracy, applicability, fitness, or completeness of the contents of this eBook. The information provided in this book is provided on an “As Is” basis, without warranty and is strictly for educational purposes. Therefore, if you wish to apply the ideas contained in this EBook, you take full responsibility for your actions.

The author and publisher disclaim any warranties (expressed or implied), merchantability, or fitness for any particular purpose. The author and publisher shall in no event be held liable to any party for any direct, indirect, punitive, special, incidental or other consequential damages arising directly or indirectly from any use of this material, which is provided “as is”, and without warranties.

The author and publisher do not warrant the performance, effectiveness or applicability of any sites listed or linked to in this eBook. All links are for information purposes only and are not warranted for content, accuracy or any other implied or explicit purpose.

Table of Contents

[Acknowledgements](#)

[Introduction](#)

[Chapter 1. The 'LAMP' stack](#)

[Chapter 2. Apache Handlers & PHP SAPI](#)

[Chapter 3. Install & Configure PHP-FPM](#)

[Chapter 4. Boost Performance with Opcode Cache](#)

[Chapter 5. Securing Apache](#)

[Chapter 6. Securing PHP \(php.ini\)](#)

[Chapter 7. Performance Tuning](#)

[Chapter 8: Installing Nginx server as Reverse-Proxy](#)

[Summary: Book Errata & Support](#)

[My Other Kindle Books](#)

[Appendix A: An Overview of Apache MPM](#)

[Appendix B: Apache 2.4 vs Apache 2.2](#)

[Appendix C: Apache's Directives](#)

Introduction

The traditional/typical way of running PHP on the webserver is via 'mod_php' (or 'mod_suphp') - which is essentially running the PHP-engine within the Apache webserver itself. It requires the least configuration and it basically just "work out of the box".

(Note: In this book, I'll use the term 'webserver' to denote the Apache httpd server unless specifically stated otherwise)

So, why use PHP-FPM if mod_php (mod_suphp) works without any fuss? What are the advantages of PHP-FPM?

There are many, but they can be boiled down to these 3 main benefits:

- **Better Memory management**
- **Better Isolation and Security:** Apache and PHP-FPM are independent of each other.
- **Scalability** - you can run Apache and PHP-FPM on separate servers.

So, what is PHP-FPM? It is basically PHP's implementation of the FastCGI application/protocol. The "FPM" stands for "**F**astCGI **P**rocess **M**anager".

Essentially, the PHP engine has its own "process manager (pm)" to manage the php processes on its own, independent of the Apache webserver.

What this means is that if you ever need to scale up quickly, you can easily do it by running the Apache webserver and PHP-FPM on separate servers. You can have a single Apache webserver on the front-end and multiple PHP-FPM servers on the backend. Apache will then redirect requests to the multiple backend PHP-FPM servers.

Ok, enough chatter about the virtues of PHP-FPM. Let me give you a quick overview of what this book covers and what you will learn from it.

In this book, we will build and configure 2 fictitious hosting accounts/domains : 'ponyvps.com' and 'donkeyvps.com'. The examples given will be based on these 2 fictitious domains.

All the sample configuration files will be given so that you can create a similar setup on your PC/Laptop or your VPS server.

What you'll need to follow along:

If you want to follow along the various configuration steps, you will need either:

1. A VPS Server or
2. A Laptop/PC running Linux (CentOS)

You can get a 1GB RAM VPS quite cheaply - between USD \$5 - \$10/mth. Some hosting companies provide yearly plans which will be even cheaper. I will provide a resource page in the Summary chapter where you can check out some of the VPS offerings by various hosting companies.

Here's a quick overview of the chapters in this book:

Introduction : You're reading this now.

Chapter 1. The 'LAMP' stack

The 'LAMP' is the acronym for 'Linux', 'Apache', 'MySQL' and 'PHP'. Nearly all modern web servers will have the LAMP stack. In this chapter, I'll provide a quick overview of each of these software components. This is a short chapter which you can skip if you already know about it.

Chapter 2. Understanding Apache Handlers / PHP SAPI

This chapter is important as it provides the background knowledge on the various PHP SAPI (Server API) and how it interfaces with Apache via the 'Handlers'. Knowing this info will help you understand and makes configuring Apache + PHP-FPM much easier.

Chapter 3. Installing & Configuring Apache + php-fpm in CentOS

This is the meat and potatoes of the book. We will setup PHP-FPM on a CentOS from scratch. We'll look at the various configuration files related to PHP-FPM and Apache - (httpd.conf, php.ini, php-fpm.conf, etc).

Chapter 4. Boost Performance with Opcode Cache

PHP is an 'interpreted' language, in other words, each piece of PHP code needs to be parsed and interpreted by the PHP engine before it can display the results. Opcode Caches are performance enhancing extensions for PHP.

In this chapter, we will look at the Zend Opcode since it has been adopted by PHP as its official opcode cache engine. The other 2 popular opcode caches are the Xcache and the APC (Alternative PHP Cache).

Chapter 5. Securing Apache

The default Apache configuration shipped by your distro is quite 'permissive' - i.e. the security settings are quite lax to ensure that the Apache server will 'work' out of the box. In

this chapter, we'll look at some of the ways we can secure our Apache server from hackers.

Chapter 6. Securing PHP (php.ini)

Although PHP is an interpreted language, it is quite powerful. You can use PHP to run system calls/commands to list files, delete files, etc. The default PHP settings allow a web-user to create a simple php page (3 lines of code) to view your server's /etc/password file or access another user's private files. **Unbelievable? I will show you how it can be done.**

Chapter 7. Performance Tuning & Benchmark

In this chapter, we'll look at various ways to tune Apache, PHP (php.ini) and php-fpm (php-fpm.conf) for maximum performance. We will use the Apache Bench ("ab") software tool to simulate real-world traffic/load to your server by sending N number of concurrent requests to your server.

Chapter 8: Installing Nginx server as Reverse-Proxy

Nginx is an upcoming webserver that is gaining popularity with many high-traffic sites. You can run to replace Apache (as a normal Web server) or you can run it as a 'Reverse Proxy' server - often used to boost the performance of high-traffic websites on Apache.

According to a recent Netcraft survey, Nginx served or proxied nearly 18% of the world's busiest website in April 2014. This chapter will guide you through the setting up of Nginx as the reverse proxy to speed up our Apache powered website

Chapter 9: Conclusion & Summary

A quick wrap-up on what we've learnt thus far. Links to external sources will be provided in this chapter for those

who want to dig deeper.

Appendix A: Overview of Apache's MPM

Appendix B: Apache 2.2 vs 2.4

Appendix C: Important Apache's Directives

NOTE: This is not a technical manual, but a practical guide that you can follow along. The materials covered in this book will give you a solid understanding on the subject matter. If you wish to go deeper, you'll have the sufficient knowledge to dig into the technical docs at the official websites of Apache, PHP, MySQL and Nginx.

The Goal of this Book :

After completing this book:

- You will have a clear understanding of the Apache Handlers: DSO, CGI/suPHP, FastCGI & PHP-FPM.
- You will know how to install and configure Apache + php-fpm + Zend OpCache on CentOS.
- You will know how to install Nginx server as the reverse-proxy to boost performance of your site.
- You will know how to Secure Apache and lock down PHP so that your users cannot use it to view sensitive system file such as the /etc/passwd.

Assumptions :

I've made some assumptions about you, the reader:

- You have a basic knowledge of Linux commands such as 'cp', 'ls', 'ln', etc 'cp', etc
- You know how to use an editor such as 'vi' or 'nano' or 'joe' to edit simple text files.

- You want to setup Apache + PHP-FPM, but you may have hit a brick-wall.
- You're looking for a hands-on, practical guide to help you get started quickly.

No prior knowledge of the Apache webserver or PHP Server API (SAPI) is required. I will explain the differences between the various “Apache Handlers/PHP SAPIs” (e.g. DSO, CGI, FastCGI, mod_suPHP) and how PHP-FPM fits into the picture.

All the necessary commands to set up the various software and configuration will be provided, with explanation. You just need to know how to type these commands on the Linux shell. I do recommend reading Chapter 2 to get a better understanding of the various Apache Handlers and PHP Server APIs (SAPI).

Chapter 1. The 'LAMP' stack

The 'LAMP' is the acronym for 'Linux', 'Apache', 'MySQL' and 'PHP'. Most of the websites on the Internet today runs on this software combination (or “stack”). This chapter will give you a quick overview of each of these software components. (note: Some might say that the 'M' refers to MariaDB – which is essentially a binary compatible “fork” from the original MySQL.)

LINUX

Linux is the 'kernel' (the heart) of the Operating System. A Linux 'distro' (distribution) such as CentOS, Ubuntu, Debian, Mint, etc includes the Linux 'kernel' PLUS an assortment of essential software packages (such as GUI/Windows system, File-Manager, etc) to make the operating system complete and useful for the end-user and system administrators.

Each 'distro' will have its own software package manager where you can install/remove and update the software packages on your server. In CentOS (RHEL), the command line package manager is called 'yum'.

APACHE

Apache one of the oldest webserver on the Internet, and is still the most popular webserver in the world today. It is still being actively maintained and developed, and it is still the market leader as reported by www.netcraft.com.

The Apache webserver (HTTP server) is modular. It has a small core and many modules which can be 'loaded' at runtime to provide various functionalities as required. Because of this architecture, the Apache webserver can be very lean and fast (with minimal modules loaded) or it can be 'bloated' with many unnecessary modules loaded.

MySQL

The 'M' in the LAMP stack is the venerable MySQL server. The MySQL server is arguably the most popular open-source relational database before it was bought over by Sun and later by Oracle.

Today, there are many people who prefer the 'MariaDB' (a binary compatible, drop-in replacement for the MySQL server) as for their database server. We won't be spending much time on MySQL as it is a huge topic by itself. I will show you how to install the MySQL server via the 'yum' (or 'apt-get') command to install MySQL if it hasn't already been installed.

(note: MariaDB is developed by the original author of MySQL - Michael Widenius 'Monty' and some of the original core group of MySQL developers. It is shipped by default in CentOS 7 instead of MySQL)

PHP

PHP is one of the most popular scripting language on the web today. It is fast and easy to learn and has many die-hard fans. Many of the world's popular Blogs and CMS (Content Management Systems) such as Wordpress, Drupal and Joomla are built on PHP.

PHP is an 'interpreted' language, i.e. PHP scripts (codes) are in plain-text and human readable. When a browser views a php page, the webserver has to send the request to the PHP engine to interpret/execute the codes and return the results back to the Apache webserver. The webserver then displays the results in 'html' to the user (webbrowser).

This 2-way communication between the webserver and PHP engine is done via the PHP SAPI (Server API) and Apache

Handler. We will discuss this in greater detail in the next chapter.

Chapter 2. Apache Handlers & PHP SAPI

This chapter is important as it provides a solid background to help you understand the Apache Handler and the PHP SAPI (Server API). This information will be very useful when you start configuring Apache + PHP-FPM later.

A webserver's role is to serve pages to the user (web-browser) when requested. In the good old days, these pages are plain static html/css/image files. You can say that these 'pages' are 'dumb' and does not change at all. It cannot access databases or retrieve data from databases. To overcome these shortcomings, software applications were created to access databases in the backend and send the results to the Apache webserver to be displayed.

These software applications can be written in any language, e.g. C/C++, Java, Python, Ruby, and of course, PHP. ***These software applications communicate with Apache via the Server API, or "SAPI" for short.***

The PHP SAPI is the module/code that provides the interface between the PHP engine and Apache. It enables Apache to communicate with the PHP engine to execute php code.

The Apache Handler is an internal directive that associates a particular file type, e.g. *.php files (files with .php extension) with an appropriate "action" to be taken - e.g. passing the request to an external FastCGI program such as the 'PHP-FPM' or to process the file itself.

For example, if a browser requests a php script (e.g. "hello.php"), Apache will then consult its "handlers" to see how to process the *.php file.

If Apache sees this directive: *AddHandler php-fpm .php*
Then it will the request to the “php-fpm” handler to process the “hello.php” file.

This “php-fpm” handler will then communicate with the PHP engine via the SAPI (Server API) to process the hello.php script and send the results to Apache to be displayed on the browser.

What’s the difference between PHP SAPI and Apache Handler?

The PHP SAPI (which we will discussed at length after this) is the “communication module/interface” that allows Apache to communicate with the PHP engine.

The Apache Handler is "an internal representation of an action” to be performed when a particular file type is requested by the web-browser.

In other words, a 'Handler' tells Apache what to do when it encounters different file types. Handlers can either be 'built-in' or included in a module. The standard built-in Apache handlers are:

- default-handler (core)
- send-as-is (mod_asis)
- cgi-script (mod_cgi)
- imap-file (mod_imagemap)
- server-info (mod_info)
- server-status (mod_status)
- type-map (mod_negotiation)

Generally, files have implicit handlers, based on the file type. Most files are served by Apache itself, but scripts such

as php, perl, python, etc have their own specific handlers because Apache needs to communicate with those 'external' application servers.

Example: If a browser requests a html page with images, Apache will handle that request itself. But if a browser requests for a PHP file (e.g. hello.php), then a suitable Handler will tell Apache how to communicate with the PHP engine to process it via the PHP SAPI.

Examples:

If you are using the mod_php (DSO) SAPI, then use this handler:

*AddHandler **php5-script** .php*

This will instruct Apache to handle files with .php extensions via the 'php5-script' handler.

If you are using the FastCGI SAPI, then you could use this handler:

*AddHandler **php-fpm** .php*

This will instruct Apache to handle .php files via the "php-fpm" handler. This "php-fpm" is a custom defined handler which we will discuss in more detail in the later chapters.

We could have defined the handler as 'fpm-engine' instead of 'php-fpm', i.e.

*AddHandler **fpm-engine** .php*

-----sidebar-----

There is another similar Apache directive called 'AddType' which is very similar to the 'AddHandler' directive. The difference is subtle, i.e. 'AddType' directive **tells the browser** (the client) how to handle a particular type of file,

e.g. what type of content to expect, for example, if it's an image file, then display it. If it's a .zip file, then download it.

'AddHandler' directive **tells Apache (the server)** how to handle a particular type of file. E.g. if the file's extension is .php, then communicate with PHP (via the PHP SAPI) to process it.

For more info about Apache Handlers:
<https://httpd.apache.org/docs/2.2/handler.html>

Ok, now that we know what is an Apache Handler, let's look at the different PHP SAPI now. There are 4 different types of PHP SAPI where Apache communicates with the PHP engine:

- mod_php (also known as DSO or Apache 2.0 Handler)
- CGI
- suPHP
- FastCGI

Note: We instruct Apache which PHP SAPI to use by using the 'LoadModule' directive in the "httpd.conf" file to load the appropriate PHP SAPI module (one of the 4 SAPIs above).

A couple of examples:

LoadModule php5_module modules/libphp5.so ← tells Apache to use 'mod_php'

LoadModule fastcgi_module modules/mod_fastcgi.so ← tells Apache to use 'FastCgi'

1. DSO (aka Apache 2.0 Handler) or 'mod_php'

This is the default PHP SAPI that gets installed when you install PHP on your server. It is also the 'simplest' because

this module loads the entire PHP engine/interpreter into Apache itself. In other words, each Apache child-process has its own dedicated PHP engine within itself. Because of this, there is minimal (if any) configuration required - PHP will just 'work' out of the box.

But, it has 2 major drawback - one, if you plan to offer hosting services to the general public, using ***mod_php*** is bad news for security because each Apache child-process will be running under the same Apache user. In other words, it is possible for User 'Fred' to access User Jim's files and vice-versa.

The other major drawback is performance. If Apache is serving mainly static content (e.g. images, CSS, Javascripts), each child-process will still load the entire PHP engine, even though there is no PHP code to process. This causes the Apache child-processes to be 'bloated' and consumes more memory than necessary.

In high-traffic, media-rich websites, this will cause serious performance issues if the server does not have sufficient memory (RAM) due to the 'bloated' Apache child-processes.

Ideally, an Apache child-process (without the embedded PHP engine) should be used to serve static content, and forward PHP files to a 'dedicated/external' PHP process. We shall see how this can be done via the 'FastCGI' SAPI.

2. CGI (Common Gateway Interface)

This is the oldest method for a webserver (e.g. Apache) to communicate with an external program (e.g. PHP). CGI is a neutral protocol that allows any webserver to connect with any language interpreter (PHP, Perl, Python, etc)

In the Apache webserver, the CGI SAPI is provided via the 'mod_cgi' or the 'mod_cgid' modules. (note: both modules are enabled (loaded) by default in Apache.)

Now, instead of the entire PHP engine running inside each child process, Apache will invoke the external program (e.g. /usr/bin/php-cgi) to process the PHP script, receive the output from the CGI program (/usr/bin/php-cgi) and then display the output to the browser.

The main drawback with the CGI SAPI is that it is CPU intensive, and it is the slowest in terms of performance. Apache will invoke the php-cgi process each time upon request, and closes it after the request has been processed.

This means that each time a client requests for a php file, there is the overhead of Apache starting a new cgi process to handle that request. It is also not possible to use an Opcode Cache to boost performance because these cgi processes are not persistent - they are opened and closed by Apache once a request has been processed.

*Note: By default, the cgi scripts invoked by Apache will run as the same user as Apache (e.g. user 'nobody' or user 'apache'). You can force these cgi scripts to run under specific user/group privileges by using the 'suexec' module (mod_suexec). It is more generic compared to suPHP which is for php only.

3. suPHP (mod_suPHP)

Some folks categorize the “suPHP” sapi as a variant of the CGI sapi - this is because Apache still execute PHP as CGI - but under the username/ID of the web-hosting account. For example, let’s consider 2 different users: “fred” and “jim”. When a browser requests a PHP file belonging to “fred” - Apache will invoke the PHP cgi as the user “fred”. Similarly

when a browser requests for a file belonging to “jim” – PHP will be invoked as the user “jim”.

This is in contrast to the CGI sapi, where PHP is invoked as the main user ‘apache’ (or user ‘nobody’) which has access to BOTH Fred’s and Jim’s files. With suPHP, access is limited to the user’s files only.

If you have multiple VirtualHosts on your server, then it is good idea to use suPHP which provides greater security where each virtual host under its own userID and permission.

The suPHP SAPI consists of the ‘mod_suphp’ module and a setuid binary which is invoked by Apache to change the UID of the process running the PHP engine. This provides the environment of running php scripts under the permissions of their owners. This is the preferred SAPI used by Cpanel/WHM servers.

Note: Since the suPHP sapi is essentially a “CGI” process – i.e. it is invoked and closed for each request, we cannot use the Opcode cache to boost performance with this SAPI as well.

4. FastCGI (e.g. 'php-fpm')

FastCGI is an enhancement to the CGI protocol - a neutral protocol that enables webserver to talk/communicate with software programs. The main difference between FastCGI and CGI is that FastCGI is 'persistent' - i.e. the FastCGI processes are kept alive by the 'pm' (process manager) so that they can be reused again by client requests.

This reduces the CPU overhead in spawning a new cgi process for each client request.

This is the preferred method of running PHP with Apache - where the PHP processes are kept separate from the Apache processes. Whenever there is a request for a php script, Apache will route it to the PHP engine via the FastCGI protocol.

These *PHP processes* are known as **"FastCGI application" or "FastCGI server"** (or FastCGI daemon) because it runs in the background and separate from Apache.

----- sidebar -----

*A **FastCGI application/server** is a process that "speaks" the FastCGI protocol - it can be PHP or Python or Perl or any other program that understands the FastCGI protocol. These FastCGI applications are managed by 'pm' (Process Managers) - which can be an **external 'pm'** such as the "PHP-FPM" server or an internal Apache 'pm' (provided by mod_fcgid)*

----- end sidebar -----

In order for Apache to communicate with the FastCGI application server, we need to install either the 'mod_fastcgi' or the 'mod_fcgid' module.

What's the difference between Apache's 'mod_fastcgi' and 'mod_fcgid'?

Both these modules enable Apache to communicate with FastCGI applications, but they differ in how these FastCGI applications are managed.

Both modules have their own in-built 'process-manager' (pm), but only "mod_fastcgi" supports an external process manager ('pm') such as the PHP-FPM. When using the external 'pm', mod_fastcgi will just act as a proxy to relay

the requests to the external fastcgi application server (e.g. PHP-FPM)

In contrast, the “mod_fcgid” does not support external ‘pm’ and therefore does not support PHP-FPM’s process manager. It uses its own in-built process manager (pm) to handle the pool of php processes.

Configuring Apache to use these internal ‘process-managers’ (pm) are quite complicated as it requires various wrappers. Therefore, the recommended way run PHP-FPM is via the “mod_fastcgi” module where we can use PHP-FPM as the external ‘pm’ (process manager)

The CLI (Command Line Interface)

This CLI SAPI is not used by Apache, but I thought I'd mentioned it just to complete the discussions on the various PHP SAPIs. Most people think of PHP as the de-facto programming language on the web, , but you can also use PHP as a scripting language on the Linux command line.

Most hardcore System Admin would either program in the 'Shell' (e.g. Bash, Zsh, Korn, etc) or use the venerable Perl scripting language or Python to automate various system tasks.

But if you are expert in the PHP language, you can actually write various system automation scripts in PHP and have it executed on the shell (just like Perl or Python scripts).

If you are interested in this topic, here's a ref: <http://www.php-cli.com/php-cli-tutorial.shtml>

Let’s briefly recap what we’ve learnt.

Chapter Summary

The Apache “handler” is an internal Apache representation that tells Apache how to handle different file types, e.g. when it encounter *.php files, what should it do?

In our case, we will define a handler (e.g. “php-fpm”) that tells Apache to communicate with the external PHP engine via the “FastCGI” PHP Server API

We discussed at length the ‘FastCGI’ SAPI – which offers real separation between the Apache processes and PHP processes, i.e. one can shutdown the PHP processes and still have the Apache server up and running – and vice-versa.

The PHP FastCGI processes (application server) are managed by the “pm” (process-manager), one of which is the the PHP-FPM.

In order for Apache webserver to communicate with the PHP-FPM application server, we need to install the ‘mod_fastcgi’ module.

PHP-FPM is very efficient in handling the php processes and server resources (CPU and memory). It also allows multiple php processes to share one Opcode cache for better performance. It also enables Apache to run each VirtualHost under its own username/ID and permission without any complicated 'wrappers' via multiple php-pools.

This alone is a big plus and solves many configuration headaches!

Chapter 3. Instal & Configure PHP-FPM

The examples in this book are based on CentOS and we'll be starting from scratch, i.e. a brand new VPS server that has "nothing" but the bare-bone CentOS Operating System on it. If your server already has some of the software packages mentioned, you may skip those sections.

Note: If you see the hash symbol, i.e. **#** << do **NOT** type this; it means that you're logged in as the superuser 'root'.

If you see the dollar symbol, i.e. **\$** << it means you're logged in as a normal user.

Preparing our server/VPS

The first thing (if you haven't done it yet) upon starting your brand new CentOS VPS is to update all the system software by:

```
# yum check-update
# yum update
```

The command above will update your CentOS server with all the latest software versions and patches.

Note: We'll be using the latest version of CentOS 6 (version 6.6) and not CentOS 7 as it is more stable and robust as of this writing.

The layout of the Apache files are:

/etc/httpd/conf/

- This is the directory where the main Apache configuration file - httpd.conf is stored

/etc/httpd/conf.d/

- This directory contains all the additional configuration files that Apache will read when it starts. Usually, these files (*.conf) will contain directives to load various modules. **After** you've installed PHP, you will find "php.conf" in this directory

/etc/httpd/modules/

- This directory is a symlink to /usr/lib/httpd/modules/ which contains all the available modules.

/etc/httpd/logs

- This directory is a symlink to /var/log/httpd/ which stores the Apache "error_log" and "access_log"

/etc/httpd/run/

- This directory is a symlink to /var/run/httpd which stores the process ID (pid) of the parent Apache process.

The Apache binaries are stored in the /usr/sbin/ directory. There are actually a few binaries:

- httpd - this is the default binary which uses the "prefork" MPM as the main engine
- httpd.worker - this is the alternative engine that uses the "worker" engine
- httpd.event - this is the newest Apache MPM engine - considered 'experimental' in Apache 2.2, but has achieved "stable" status in Apache 2.4.

Let's check the default modules that comes with CentOS.

```
# cd /usr/lib/httpd/modules && ls
```

*Note: if you have installed 64-bit of CentOS, the path will be:

```
# cd /usr/lib64/httpd/modules && ls
```

You will see a file called 'mod_cgi.so'. This is the CGI SAPI module. We will need to install another module called 'mod_fastcgi.so' from the "RepoForge" repository by adding it into our list of software repositories.

Installing the required Software Repositories:

We will need to install software packages from the following repositories:

- EPEL
- REMI
- RepoForge (formerly RPMForge)

Before we begin, we need to determine whether your server/vps has a 32-bit or 64-bit CPU.

CPU Architecture: 32-bit or 64bit ?

```
# cat /proc/cpuinfo | grep flags
```

You will see a string of parameters, e.g.

```
flags      : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca c
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb
scp lm constant_tsc arch_perfmon pebs bts rep_good xtopology nonstop_tsc ape
erf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 fma cx16 xtpr
pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16
rand lahf_lm abm arat epb xsaveopt pln pts dts tpr_shadow vnmi flexpriority
vpid fsgsbase bmi1_hle avx2 smep bmi2 erms invpcid rtm
```

If you see 'lm' (long mode) among those parameters, then your CPU is 64-bit. A 64-bit CPU can run both 64-bit or a 32-bit OS. If your CPU is 32-bit, then you can only run a 32-bit OS.

OS (Operating System): 32-bit or 64-bit ?

```
# getconf LONG_BIT
```

If it shows 64, then your OS is 64-bit. If it shows 32-bit, then your OS is 32-bit.

You can also use this command: `# uname -a`

If it shows something like: "2.6.32-042stab084.20 x86_64 x86_64 x86_64 GNU/Linux"

The "**x86_64**" tells you that your OS is 64-bit.

If it shows something like: "2.6.18-028stab101.1 i686 i686 i386 GNU/Linux"

The "**i686 i386**" means 32-bit OS.

TIP: when determining which software version to install, you only need to know whether your OS is 32-bit or 64-bit. If your OS is 32-bit, then install the 32-bit version.

Install the EPEL Repository:

At the command line, run this command:

```
# yum install epel-release
```

For more information about the EPEL repository, visit:

<https://fedoraproject.org/wiki/EPEL>

(We'll be installing the Zend Opcode Cache software from this EPEL repository)

Install the REMI repository:

Run the following commands to download and install the REMI repository:

```
# wget http://rpms.famillecollet.com/enterprise/remi-release-6.rpm
```

```
# rpm -ivh remi-release-6.rpm
```

For more info about the REMI repo, visit:

<http://blog.famillecollet.com/pages/Config-en>

The REMI software repository a yum repo maintained by a French programmer - Remi Collet. It usually has the latest version of the software in found in the core CentOS repositories, and it uses the same package names as in the official repositories.

Note: You'll have to explicitly 'enable' this repo if you want to install a particular software from this repo. The Remi repository requires that we install the EPEL repository first.

We will use this REMI repo to install MySQL 5.5 and also PHP 5.4. The CentOS 6 comes with MySQL 5.1.73 and PHP 5.3 (which has reached its end of life cycle).

Install the RepoForge Repository

Launch your web browser and go to:
<http://repoforge.org/use/>

You will see:

EL 7: x86_64

EL 6: i686, x86_64

EL 5: i386, x86_64, ppc

EL 4: i386, x86_64, ppc

EL 3: i386, x86_64

*note: EL = Enterprise Linux (same as 'CentOS'). So, EL 6 = CentOS 6

i386, i686 = For 32-bit OS.

x86_64 = For 64-bit OS.

CentOS 7 (RedHat Enterprise Linux 7) no longer supports 32-bit architecture. If your VPS has only 1GB RAM, a 32-bit CentOS 6.6 will be more resource friendly - thus giving your server more resources (CPU & Memory) for your applications.

Download the appropriate package for your OS, e.g.

For CentOS 6, 32-bit:

```
# wget http://pkgs.repoforge.org/rpmforge-release/rpmforge-release-0.5.3-1.el6.rf.i686.rpm
```

For CentOS 6, 64-bit:

```
# wget http://pkgs.repoforge.org/rpmforge-release/rpmforge-release-0.5.3-1.el6.rf.x86\_64.rpm
```

Now, let's install it:

```
# rpm -ivh rpmforge-release-0.5.3-1.el6.rf.i686.rpm
```

Now that we have installed the required software repositories (RepoForge, EPEL & REMI), let's install the 'mod_fastcgi' module.

Install "mod_fastcgi"

```
# yum install mod_fastcgi
```

You will see the following on your screen.

```
=====
Package                Arch          Version      Repository
=====
Installing:
mod_fastcgi            i686          2.4.6-2.el6.rf  rpmforge

Transaction Summary
=====
Install      1 Package(s)

Total download size: 112 k
Installed size: 326 k
Is this ok [y/N]: y
```

Enter 'y' and hit the enter key. It will install mod_fastcgi on your server.

Let's verify it by going to the /usr/lib/httpd/modules directory.

```
# cd /usr/lib/httpd/modules && ls mod_fastcgi*
```

You will see the file 'mod_fastcgi.so' (which wasn't there earlier).

```
# cd /etc/httpd/conf.d && ls
```

You will also see the file 'fastcgi.conf' which did not exist earlier.

Note: We will modify this file later to enable it to communicate with php-fpm process manager.

Install MySQL

Before installing PHP, please check that MySQL is already installed on your server. Otherwise, the PHP installation of 'php-mysql' will fail. A freshly provisioned CentOS VPS does not have MySQL pre-installed.

To install MySQL, enter:

```
# yum --enablerepo=remi install mysql mysql-server
```

```
=====
Package                Arch          Version              Repository
=====
Installing:
mysql                  i686          5.5.40-1.el6.remi    remi
mysql-server           i686          5.5.40-1.el6.remi    remi
Installing for dependencies:
compat-mysql51         i686          5.1.54-1.el6.remi    remi
libaio                 i686          0.3.107-10.el6       base
mysql-libs             i686          5.5.40-1.el6.remi    remi
perl-DBD-MySQL         i686          4.013-3.el6          base

Transaction Summary
=====
Install      6 Package(s)

Total download size: 18 M
Installed size: 82 M
Is this ok [y/N]: █
```

This will install MySQL server and its dependencies. If this is your first time installing MySQL, you need to start the service and run the 'mysql_secure_installation' to secure it.

Note the extra parameter: `--enablerepo=remi` which tells the yum utility to use the “remi” repository for this install instead of the default base repos.

Once it's installed, start the MySQL server and run the 'secure installation' script to secure it. follow the prompt on your screen - enter 'Y' to accept the recommended settings. Choose a strong password!

```
# service mysqld start && mysql_secure_installation
```

Install PHP / PHP-FPM

The default CentOS does not have PHP pre-installed. Type this command at the shell:

```
# yum install --enablerepo=remi php php-cli php-common php-gd php-mbstring php-mcrypt php-mysql php-fpm php-pear
```

Package	Arch	Version	Repos
Installing:			
php	i686	5.4.34-1.el6.remi	remi
php-cli	i686	5.4.34-1.el6.remi	remi
php-common	i686	5.4.34-1.el6.remi	remi
php-fpm	i686	5.4.34-1.el6.remi	remi
php-gd	i686	5.4.34-1.el6.remi	remi
php-mbstring	i686	5.4.34-1.el6.remi	remi
php-mysql	i686	5.4.34-1.el6.remi	remi
php-pear	noarch	1:1.9.5-3.el6.remi	remi
Installing for dependencies:			
freetype	i686	2.3.11-14.el6_3.1	base
libX11	i686	1.6.0-2.2.el6	base
libX11-common	noarch	1.6.0-2.2.el6	base
libXau	i686	1.0.6-4.el6	base
libXpm	i686	3.5.10-2.el6	base
libxcb	i686	1.9.1-2.el6	base
libxslt	i686	1.1.26-2.el6_3.1	base
php-pdo	i686	5.4.34-1.el6.remi	remi
php-process	i686	5.4.34-1.el6.remi	remi
php-xml	i686	5.4.34-1.el6.remi	remi

Enter “y” when prompted. The command above will install a bunch of PHP software – including “php-fpm” and its associated dependencies/libraries. The location of the files:

/usr/bin/php-cgi

- This is the “cgi-fcgi” binary that will be called if we use the 'CGI' sapi or the built-in Apache's fcgid module.

/usr/sbin/php-fpm

- This is the PHP FastCGI server binary that communicates with Apache via the ‘mod_fastcgi’ module. It runs as a server process with its own ‘pm’ (process manager) to manage each PHP child-processes.

Note that it is in the /usr/**sbin** and not /usr/bin

The main PHP configuration file is /etc/php.ini, and additional configuration files are stored in the /etc/php.d/ directory – these are simple text files that instruct PHP what modules/extensions to load at run time, e.g. “mysql.ini” file.

The main PHP-FPM configuration file is /etc/php-fpm.conf.

The individual PHP-FPM “pool” files are stored in the /etc/php-fpm.d/.

Each virtual host user will have its own unique pool file.

We will use the provided default sample pool.conf file later in this chapter.

Understanding the Apache Startup Process.

When the Apache server starts, it will first read the main config file - /etc/httpd/conf/httpd.conf and load all the predefined modules in that file.

In addition, it will also scan the /etc/httpd/conf.d/ directory for any *.conf file and load the directives/modules specified

on those *.conf files.

There are 2 important *.conf files in the /etc/httpd/conf.d/ directory

- **php.conf** - which tells Apache to load the default DSO handler to process php files. This is simple and straightforward. No additional configuration is needed if you plan to use the DSO module.
- **fastcgi.conf** which tells Apache to load the 'fastcgi' module to communicate with an external FastCGI application server. We will replace the default file content in fastcgi.conf with our own code/directives.

Apache will load both modules (DSO) and FastCGI if both *.conf file exist in this directory.

If the mod_php (DSO) module is loaded, then Apache will use it instead of the "mod_fastcgi" which is what we want. To prevent the mod_php from being loaded, we will rename the file extension from ".conf" to ".inc".

```
# cd /etc/httpd/conf.d
# mv php.conf dso.inc
# mv fastcgi.conf fastcgi.inc
```

Then, we create a symbolic link (symlink) to the appropriate configuration file:

If we want to run Apache with the DSO module, we create a symlink to "dso.inc"

```
# ln -s dso.inc php.conf
```

If we want to run Apache with the FastCGI module, we create a symlink to the "fastcgi.inc"

```
# ln -s fastcgi.in php.conf
```

```
[root@srv5 conf.d]# ln -s fastcgi.inc php.conf
[root@srv5 conf.d]# ls -l
total 16
-rw-r--r-- 1 root root 990 Oct 16 04:12 dso.inc
-rw-r--r-- 1 root root 1510 Jun 8 2012 fastcgi.inc
lrwxrwxrwx 1 root root 11 Nov 13 10:55 php.conf -> fastcgi.inc
-rw-r--r-- 1 root root 392 Oct 16 10:46 README
-rw-r--r-- 1 root root 299 Aug 15 02:57 welcome.conf
[root@srv5 conf.d]#
```

Now that all the software has been installed, let's setup the configuration files now.

(A) Apache Configuration:

1. */etc/httpd/conf.d/fastcgi.inc*

The first file you need to edit is the 'fastcgi.inc' file. (It was originally "fastcgi.conf" - but we have renamed it to "fastcgi.inc" so that we can selectively load either the fastcgi or the dso module via the appropriate symlink).

Open that file with your favorite editor.

Remove all the contents in this file and paste the following snippet:

```
## --- content of fastcgi.conf ---##
Loadmodule fastcgi_module modules/mod_fastcgi.so
AddHandler php-fpm .php
Action php-fpm /php-fpm virtual
```

The 1st line tells Apache to load the 'FastCGI' module so that it can communicate with the 'php-fpm' process manager.

The 2nd line, "**AddHandler** *php-fpm* **.php**" registers the 'php-fpm' as the custom Apache Handler to process all files with the .php extension.

The 3rd line: "Action *php-fpm* **/php-fpm** virtual"

The "Action" directive tells Apache what script ("/php-fpm") to run when it encounters the registered handler (php-fpm).

In other word, files with .php extension will be processed by the *virtual* "/php-fpm" script.

The modifier “virtual” tells Apache the script does not really exist, it is “virtual”.

The syntax is: Action <handler> <cgi-script> [virtual]

The '<cgi-script>' (in our example above, /php-fpm) is the URL-path to the resource that has been designated as the FastCGI script using the 'AddHandler' directive.

The [virtual] modifier tells Apache not to check if the specified file exists.

2. /etc/httpd/conf/httpd.conf

Let's turn our attention to the main Apache's configuration file and create 2 fictitious domains as VirtualHosts, i.e. *donkeyvps.com* & *ponyvps.com*

We'll be using these 2 fictitious domains in our discussion for the rest of this book.

TIP: You can actually create these 2 fictitious domains and access them live on your VPS server on the Internet. All you need to do is to manually enter the IP addresses of your server in your "hosts" file. For example, if your PC/laptop is running CentOS or Ubuntu (or another Linux distro), you can edit the file: /etc/hosts with these domains, e.g.

X.X.X.X ponyvps.com donkeyvps.com

where 'X.X.X.X' is the IP address of your VPS/server

If your PC/laptop is a Windows machine, the “hosts” file is located at:
C:\windows\system32\drivers\etc\hosts

(ref: <http://helpdeskgeek.com/windows-7/windows-7-hosts-file/>)

The first step is to create a unique 'web-user' account for each of these virtualhost. The reason we do this is because we want to create a unique php-fpm process pool for these users – so that the php processes for “donkeyvps.com” will run as the user “donkey” and the php processes for the “ponyvps.com” will run as the user “pony”. This will provide

the security and isolation for each virtual host on your server.

```
# useradd donkey --shell=/sbin/nologin
# useradd pony --shell=/sbin/nologin
```

The 2 commands above will add the user 'donkey' and user 'pony' with no login privileges. Now, let's create the directories for these 2 virtual hosts to store the webpages.

```
# cd /home/donkey && mkdir www cgi-bin logs
# cd /home/pony && mkdir www cgi-bin logs
```

And create a simple 'index.php' file for each of these 2 sites. It has only 1 line of code:

```
<?php phpinfo(); ?>
```

Save it as 'index.php' and store it at /home/ponywww/ and /home/donkey/www/.

Change ownership and permission of these directories to the user 'donkey' and 'pony'

```
# cd /home
# chown -R donkey:donkey donkey && chmod 755 donkey
# chown -R pony:pony pony && chmod 755 pony
```

At this point in time, the Apache server is not aware of these 2 virtual hosts yet because we've not set them in the 'httpd.conf' file. So let's do that now.

Open the '/etc/httpd/conf/httpd.conf' file with your editor and do the following:

Look for this line: *DirectoryIndex index.html index.html.var*
Change it to: *DirectoryIndex index.html **index.php***

Then, go to the end of the file and insert the following code snippets:

(you can download/copy the code snippets at <http://ilovevirtualmin.com/book-3/>)

For VirtualHost: DonkeyVPS.com

```
NameVirtualHost *:80

<VirtualHost *:80>
    ServerAdmin webmaster@donkeyvps.com
    DocumentRoot /home/donkey/www
    ServerName donkeyvps.com
    ErrorLog /home/donkey/logs/error_log
    CustomLog /home/donkey/access_log common
<IfModule mod_fastcgi.c>
    Alias /php-fpm /home/donkey/cgi-bin/php-fpm
    FastCgiExternalServer /home/donkey/cgi-bin/php-fpm -socket /home/donkey/cgi-
bin/donkey.sock -pass-header Authorization -idle-timeout 100
</IfModule>
</VirtualHost>
```

Take note: Just before the <VirtualHost *:80> directive, make sure there is a line that says:

```
NameVirtualHost *:80
```

The line above is very important, it tells Apache to use the Name-based virtual hosts.

For VirtualHost: PonyVPS.com

```
<VirtualHost *:80>
    ServerAdmin webmaster@ponyvps.com
    DocumentRoot /home/pony/www
    ServerName ponyvps.com
    ErrorLog /home/pony/logs/error_log
    CustomLog /home/pony/logs/access_log common
<IfModule mod_fastcgi.c>
    Alias /php-fpm /home/pony/cgi-bin/php-fpm
    FastCgiExternalServer /home/pony/cgi-bin/php-fpm -socket /home/pony/cgi-bin
/pony.sock -pass-header Authorization -idle-timeout 100
</IfModule>
</VirtualHost>
```

***NOTE:** You can replace the * with your server's IP address.

Take note of the "Alias" and the "FastCgiExternalServer" directives - these 2 directives are enclosed within the <IfModule mod_fastcgi.c> container and will be executed only if we are running Apache with the fastcgi module enabled.

The purpose of the “Alias” directive is to map the “/php-fpm” URI to a virtual file-path that has been designated as the ‘FastCgiExternalServer’ applicaton, i.e.

For ("donkeyvps.com"):

Alias /php-fpm /home/donkey/cgi-bin/php-fpm

For ("ponyvps.com"):

Alias /php-fpm /home/pony/cgi-bin/php-fpm

The file-paths '/home/donkey/cgi-bin/php-fpm' (and '/home/pony/cgi-bin/php-fpm-') do not exist. These are used as virtual ‘hooks’ for the “FastCgiExternalServer” directive to process the php requests. URIs that Apache resolve to this filename (via the Alias directive) will be handled by the external FastCGI application (PHP-FPM).

If you recalled, in the “/etc/httpd/conf.d/fastcgi.inc” file, we have set the 'Action' directive to: ***Action php-fpm /php-fpm virtual***

*The modifier '***virtual***' tells Apache expect virtual URI/file path.

The PHP-FPM server can accept connections via the TCP port or via the unix socket:

-host hostname:port (configured to listen via TCP port)

-socket /path/to/socketname.sock (configured to listen via the Unix socket)

You can mix and match how PHP-FPM listens for incoming connections for each different pools, for example, in the 'donkeyvps.com' virtualHost, we can set it to listend to a Unix socket, and in “ponyvps.com”, we can set it to listen to a TCP port.

Example: Listening to a Socket

FastCgiExternalServer **filename** -socket **socketname**
[option ...]

FastCgiExternalServer **/home/pony/cgi-bin/php-fpm** -socket **/home/pony/cgi-bin/pony.sock** -pass-header Authorization -idle-timeout 100

Example: Listening to a Port

FastCgiExternalServer **filename** -host **hostname:port**
[option ...]

FastCgiExternalServer **/home/pony/cgi-bin/php-fpm** -host **X.X.X.X:9001** -pass-header Authorization -idle-timeout 100

(where 'X.X.X.X' is the IP address of the server running PHP-FPM. Usually, it's 127.0.0.1 since we're using the same server)

Warning: If you have multiple virtual hosts on the same server, and you plan to use the TCP port, make sure each virtual host specify a unique port, e.g. virtual host #1 uses port 9001, virtual host #2 uses port 9002 and so on. Otherwise, you will encounter errors.

What are the Pro and Cons of using Unix Sockets or TCP ports?

Unix sockets is generally faster (provides better performance) because it does not have the associated TCP/network overhead, but it is limited to the resources on the local server (localhost).

But if you need to scale out, you could have multiple dedicated servers running the PHP-FPM application server, and have each site point to a different PHP-FPM server, e.g.

FastCgiExternalServer **/home/pony/cgi-bin/php-fpm** -host **192.168.1.10:9000**

FastCgiExternalServer **/home/pony/cgi-bin/php-fpm** -host **192.168.1.11:9000**

FastCgiExternalServer **/home/pony/cgi-bin/php-fpm** -host **192.168.1.12:9000**

OR, insert a load-balancer in front of these PHP-FPM application servers and have it load-balance your web-application among these servers. This allows you to scale your website based on demand. Ok, now that the Apache's configuration is all set, let's look at PHP-FPM.

(B) PHP-FPM Configuration:

The main PHP-FPM configuration file is `/etc/php-fpm.conf` – which has lots of helpful comments. For now, make sure the file contains the following code snippets:

```
-----  
[global]  
pid = /var/run/php-fpm/php-fpm.pid  
error_log = /var/log/php-fpm/error.log  
emergency_restart_threshold = 10  
emergency_restart_interval = 1m  
process_control_timeout = 15s  
daemonize = yes  
;; pool definitions ;;  
include=/etc/php-fpm.d/*.conf  
-----
```

The more important files are the 'php-fpm' pool files located in the `/etc/php-fpm.d/` directory. Each user (VirtualHost) should have its own 'pool' file that defines its own PHP-FPM child processes.

Create a file called “donkey.conf” in the `/etc/php-fpm.d/` directory.

[donkey]

```
user = donkey  
group = donkey  
listen = /home/donkey/cgi-bin/donkey.sock  
listen.allowed_clients = 127.0.0.1  
listen.owner = donkey  
listen.group = apache  
listen.mode = 0660  
pm = dynamic  
pm.max_children = 30  
pm.start_servers = 5  
pm.min_spare_servers = 3  
pm.max_spare_servers = 15
```


Create a file called “pony.conf” in the /etc/php-fpm.d/ directory.

[pony]

```
user = pony
group = pony
listen = /home/pony/cgi-bin/pony.sock
listen.allowed_clients = 127.0.0.1
listen.owner = pony
listen.group = apache
listen.mode = 0660
pm = dynamic
pm.max_children = 30
pm.start_servers = 5
pm.min_spare_servers = 3
pm.max_spare_servers = 15
```

These 2 files (donkey.conf and pony.conf) tells the PHP-FPM server to start child-processes under these 2 userIDs and handles incoming requests from the respective Virtualhosts.

Also, note the following:

```
listen.group = apache
listen.mode = 0660
```

On our server, the Apache web-server runs as user “apache” - hence the “listen.group” is set to “apache”. (In Ubuntu, the Apache web-server runs as the user “www-data”, and the listen.group would then be set to 'www-data')

NOTE: Prior to PHP version 5.3.38 OR 5.4.28 OR 5.5.11 - the default listen.mode is “0666” which is “world-writable” to the unix socket. This is a huge security hole because it allows a skilled attacker to take advantage of the 'world-writable' socket.

This security bug was fixed in subsequent release of PHP, but it poses a problem for users who already have an existing working PHP-FPM and are upgrading to the latest version of PHP. These users will probably encounter this cryptic error message:

```
“(13)Permission denied: FastCGI: failed to connect to server”  
(ref: https://bugs.php.net/bug.php?id=67060)
```

The error above is caused by the existing 'pool-configuration' file - if the “listen.mode” is not explicitly set, then it will default to “0660” - which means the 'owner' and 'group' has read/write access to the socket, but not “others”. The web-server (Apache) usually runs under the user “apache” or “httpd” - and hence, fall under the “others” group which does not have access to the php-fpm socket. This causes the “failed to connect” error message.

A quick fix (as advocated by many online articles) is to manually add: “listen.mode=0666” - which defeats the purpose of the bug-fix - which was to disable the 'world-writable' bit.

A better solution is to change the “listen.group” to the userID of the Apache server, e.g. `listen.group = apache` where “apache” is the userID of the web-server.

If your Apache server is running as 'httpd', then set it to:
`listen.group = httpd`

A more secure solution would be to set the `listen.group` to be the same as the user, e.g.

```
listen.owner = pony  
listen.group = pony
```

But this will also cause permission error because the Apache does not have permission to listen/communicate via the socket. The workaround for this is to add the “Apache” user

to the “pony” group (a secondary group for Apache). For example, issue this command:

```
# cat /etc/group
```

```
mysql:x:27:  
pony:x:500:  
donkey:x:501:
```

You will see that the groups “pony” and “donkey” does not have any other 'members' except themselves. After adding the user 'apache' to these groups, via:

```
# usermod -aG pony apache  
# usermod -aG donkey apache
```

Now, if we look at the file /etc/group again, we'll see:

```
mysql:x:27:  
pony:x:500:apache  
donkey:x:501:apache
```

This adds the 'apache'

user to the secondary groups 'pony' and 'donkey' and gives the Apache user (apache) read-write access to the PHP-FPM sockets, e.g.

```
listen.mode =0660  
listen.owner=pony  
listen.group=pony (the 'apache' user is now a member of this  
'pony' group).
```

And, you can also change the directory permission to “750” which is more secure and prevents other users from snooping/viewing another user's file via PHP scripts. We'll see the implication of this in the Securing PHP chapter.

```
# chmod 750 /home/pony  
# chmod 750 /home/donkey
```

Below are the commonly used directives and their explanation:

```
listen.allowed_clients = 127.0.0.1
```

- instructs the php-fpm server to accept connections from the localhost only (127.0.0.1)

listen = 127.0.0.1:9000

- *Tells PHP-FPM to accept connections on the localhost at port '9000'. The default pool file (www.conf) uses this port. If firewall is enabled, ensure port 9001 is open*

listen = /home/pony/cgi-bin/pony.sock

- *Tells PHP-FPM to accept connections via the Unix socket. This gives better performance, but is limited to the local server, it cannot be scaled out to multiple servers*

pm.max_children = 30

- *Maximum number of PHP child processes that can be alive at the same time*

pm.start_servers = 5

- *Number of child processes created on startup*

pm.min_spare_servers = 3

- *Minimum number of child-processes in 'idle' state (waiting for connection). If the number of idle processes is less than this, then PHP-PM will spawn additional child-processes*

pm.max_spare_servers = 15

- *Maximum number of child-processes in 'idle' state (waiting for connection). If the number of idle processes is bigger than this value, it will be killed*

pm.max_requests = 500

- *This tells PHP-FPM to kill a child process after it has served 500 requests. Useful for controlling memory leak, especially if you are running Wordpress with many plugins - some of which are poorly written and can cause memory leaks*

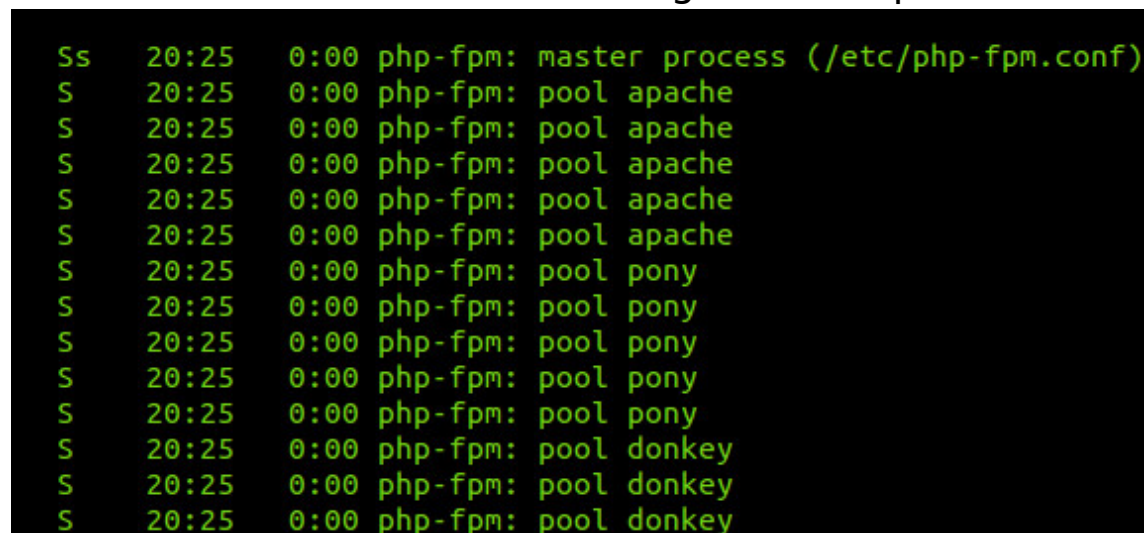
pm = <TYPE>

- *Possible values for <TYPE> are: "dynamic", "static" and "ondemand".*
- *This 'pm'(process manager) directive tells PHP-FPM how many processes should be alive. Setting it to "dynamic" offers better handling of server resources, especially memory since php-fpm processes are started only when needed instead of staying in memory all the time. The "ondemand" option, no child-process is created at startup, but only on demand*

Ok, now that we've updated/created the necessary configuration files for Apache and PHP-FPM, let's start our PHP-FPM application server.

```
# service php-fpm start
# ps aux | grep php-fpm
```

Here's a screenshot of the running PHP-FPM processes:



A screenshot of a terminal window showing the output of the 'ps aux' command. The output lists several processes, all identified as 'php-fpm'. The first line is the master process: 'php-fpm: master process (/etc/php-fpm.conf)'. Following it are multiple worker processes, grouped into pools. There are three pools named 'apache', each with five worker processes. There are three pools named 'pony', each with three worker processes. Finally, there are two pools named 'donkey', each with two worker processes. The terminal text is as follows:

```
Ss  20:25  0:00 php-fpm: master process (/etc/php-fpm.conf)
S    20:25  0:00 php-fpm: pool apache
S    20:25  0:00 php-fpm: pool apache
S    20:25  0:00 php-fpm: pool apache
S    20:25  0:00 php-fpm: pool apache
S    20:25  0:00 php-fpm: pool apache
S    20:25  0:00 php-fpm: pool pony
S    20:25  0:00 php-fpm: pool pony
S    20:25  0:00 php-fpm: pool pony
S    20:25  0:00 php-fpm: pool pony
S    20:25  0:00 php-fpm: pool pony
S    20:25  0:00 php-fpm: pool donkey
S    20:25  0:00 php-fpm: pool donkey
S    20:25  0:00 php-fpm: pool donkey
```

```
# service httpd restart
```

Now, let's point our browser to our fictitious domain - www.donkeyvps.com
(we had created a simple 'index.php' file with this code: **<? php phpinfo(); ?>**)

PHP Version 5.4.39



System	Linux srv11.ilovevirtualmin.com 2.6.32-042stab106.4 #1 SMP Fri Mar 27 15:19:28 MSK 2015 x86_64
Build Date	Mar 19 2015 06:59:53
Server API	FPM/FastCGI
Virtual Directory	disabled

You should see the “Server API” as “FPM/FastCGI”

TIP: PHP-FPM works better with the Apache 'worker' MPM compared to the 'prefork' mpm.

(Refer to Appendix A for a quick overview of Apache’s “MPM” – Multi-Processing-Module)

To use the 'worker' mpm instead of the default 'prefork', follow the steps outlined below:

```
# cd /usr/sbin
# cp httpd httpd.prefork
# rm httpd
# ln -s httpd.worker httpd
```

This will create a symbolic link (called 'httpd') to the 'httpd.worker' binary

```
lrwxrwxrwx 1 root root      12 Nov 13 11:33 httpd -> httpd.worker
-rwxr-xr-x 1 root root 356536 Oct 16 10:46 httpd.event
-rwxr-xr-x 1 root root 344080 Nov 13 11:33 httpd.prefork
-rwxr-xr-x 1 root root 356536 Oct 16 10:46 httpd.worker
```

To verify, you can issue this command: `httpd -V`

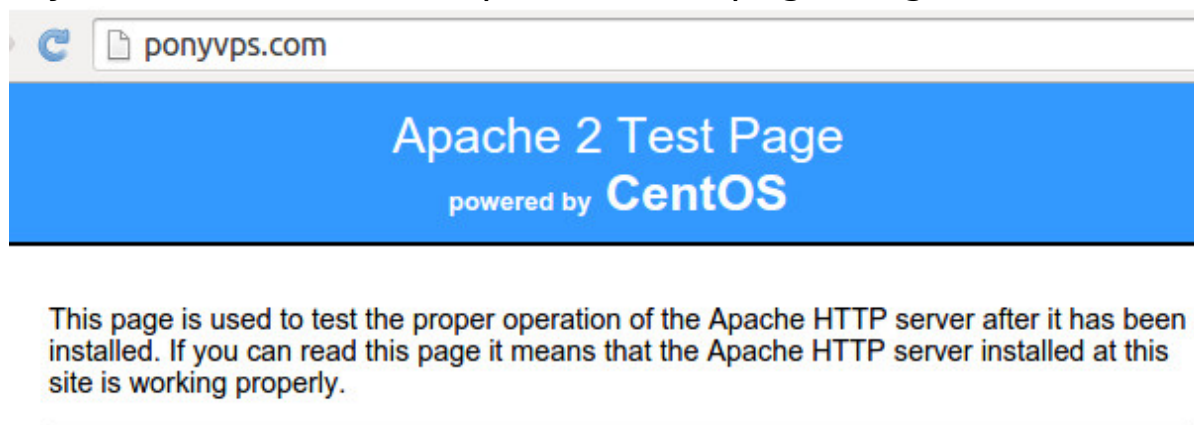
```
Architecture: 32-bit
Server MPM:   Worker
  threaded:   yes (fixed thread count)
  forked:     yes (variable process count)
Server compiled with....
  -D APACHE_MPM_DIR="server/mpm/worker"
  -D APR_HAS_SENDFILE
```

Notice “Serve MPM: Worker”. If you want to switch back to the prefork MPM, then:

```
# rm httpd && ln -s httpd.prefork httpd
```

Troubleshooting Tips:

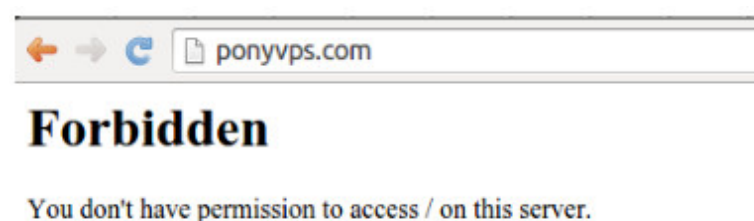
If you see the default 'Apache2 Test page', e.g.



Check the following line in /etc/httpd.conf exist:

DirectoryIndex index.html index.php

If you get “Forbidden” (Permission) Error like the screenshot below:



Check the directory permission for “/home/pony” and “/home/donkey” - the default permission when you create these users are '700' - which causes this error. You need to:

```
# chmod 755 /home/pony && chmod 755 /home/donkey (OR  
chmod 750 if you add the user 'apache' to the 'pony group'  
and 'donkey group')
```


Chapter 4. Boost Performance with Opcode Cache

Being an interpreted language, the PHP scripts (codes) need to be parsed and compiled into 'opcodes' (or 'bytecodes') before it is executed by the PHP engine ('interpreter'). The sequence of processes looks like this:

PHP code > Parse > Compile to Opcodes > Execute (ZendEngine) > Display Result

The 'parsing', 'compilation' and 'execution' is done by the PHP engine (interpreter).

Each time a web-browser requests for a PHP page/script, it has to go through the entire process (parse, compile & execute) which is time consuming.

This is where the "Opcode Cache" comes in. It is a performance enhancement extension for PHP scripts. A PHP script that has been parsed and compiled to Opcodes/Bytecodes can be cached in the Opcode cache memory. This means that the next request for that same PHP script will be executed immediately from the Opcode cache without having to go through the 'parse' and 'compile' process again.

The first opcode cache was a piece of propriety software from Zend which wasn't widely adopted. An open source alternative, aptly named 'APC' (Alternative PHP Cache) was created and became very popular. At one point, the PHP group decided to adopt the APC into the mainstream PHP engine (in the 5.4 branch), but somehow, development of the APC slowed down and lacked some cool features found in Zend's offering.

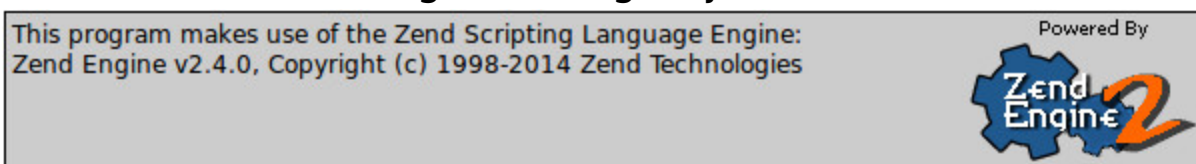
With the release of PHP 5.5, Zend decided to contribute to the PHP project and release their latest opcode cache (called the 'Zend OpCache') software under the Open Source License.

It is now included with PHP itself (PHP 5.5 onwards). It is also back-ported for the older PHP version (5.4, 5.3 and 5.2).

Another popular Opcode Cache is the "XCache" - from the developers of Lighttpd web-server. If you are interested to find out more about the XCache, here's a good starting point: <http://xcache.lighttpd.net/wiki/Faq>

In this chapter, we'll look at specifically at Zend Opcache because it has been officially adopted by the PHP group and it is compiled as a shared extension by default in PHP. It is also backported to PHP 5.4, 5.3 and 5.2. In terms of performance, it is also the fastest compared to APC and XCache from my own benchmark tests.

Before you install the Zend-Opcache, take a look at the "phpinfo();" page again, e.g. go to <http://ponyvps.com> and look for the "ZendEngine" badge - just before "PHP Credits"



Note that there is no mention of "Opcode cache"

Installing Zend-Opcache on CentOS

Installing the Zend Opcache in CentOS is easy and straightforward. The 'yum' utility will setup the necessary libraries and enable the appropriate Zend extension/modules.

```
# yum --enablerepo=remi install php-pecl-zendopcache
```

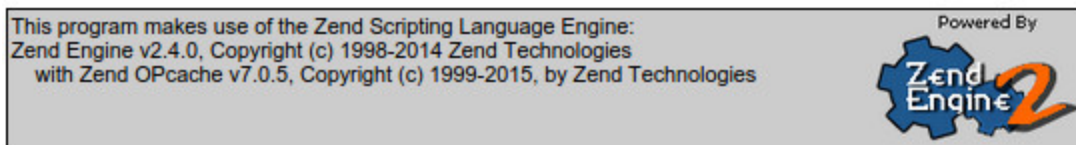
It will install the ZendOpcache and create the 'opcache.ini' in the /etc/php.d/ directory.

Once you've installed the ZendOpcache, remember to restart php-fpm!

(You don't have to restart Apache since both are independent of each other now)

Service php-fpm restart

If you view your site, you will see the Zend Opcache being mentioned.



Note: If you wish to disable the ZendOpcache for some reason, just rename the /etc/php.d/opcache.ini to /etc/php.d/opcache.ini.bak and restart php-fpm.

Chapter 5. Securing Apache

The default Apache configuration is quite 'permissive' - i.e. the default security settings are quite lax. I believe this is done on purpose to ensure that the Apache server 'works' out of the box. In this chapter, we'll look at some of the ways we can secure our Apache server.

We'll be making changes/edits to the main Apache conf file - /etc/httpd/conf/httpd.conf

1. Disable Apache Signature and Banner

One of the first thing a hacker does before attempting to hack into your server is 'information gathering' - i.e. what Operating System is running on your server, which version of Apache and which PHP version. Knowing this information will help these hackers craft their attacks or find vulnerabilities in the software versions running on your server.

The default values are:

ServerSignature On

ServerTokens OS

Following are possible ServerTokens values:

ServerTokens Prod displays "Server: Apache"

ServerTokens Major displays "Server: Apache/2"

ServerTokens Minor displays "Server: Apache/2.2"

ServerTokens Min displays "Server: Apache/2.2.15"

ServerTokens OS displays "Server: Apache/2.2.15 (CentOS)"

ServerTokens Full displays "Server: Apache/2.2.15 (CentOS) PHP/5.3.3"

(If you don't specify any ServerTokens value, this is the default)

Recommendation: Turn the 'ServerSignature Off', i.e.

ServerSignature Off

This will remove all information regarding your Apache and PHP software version.

2. Remove un-necessary Modules

Apache has a lot of modules - most of them are enabled by default because it is supposed to 'just work out of the box'. A default Apache install on a CentOS server comes preloaded with over 50+ modules to suit a variety of environments. To see the list of modules:

```
# httpd -M
```

To get a quick count of the loaded modules:

```
# httpd -M |wc -l
```

To disable these modules, you can comment out the 'LoadModules' line in the /etc/httpd/conf/httpd.conf

WARNING: Do not disable a bunch of modules at one go. If you do, Apache may fail to start because some of the directives in the configuration require certain modules to be loaded.

For example, the 'autoindex_module' is required for the directive "IndexOptions"

So if you disable the autoindex_module, you also need to disable the 'IndexOptions' directive above by commenting it out - as well as all the 'AddIcon', 'AddIconByEncoding', 'AddIconByType', 'DefaultIcon' directives.

Otherwise, Apache will complain and refuse to start. A safer method is to disable each one module (put a '#' at the beginning of the line to comment it out) and test by entering this command at the command line: `httpd -M`

If that particular module is required, Apache will complain and display an error message.

Here's a short list of modules that you can try disabling:

- `mod_dav`: provides functionality to edit documents on a remote webserver.
- `autoindex`: displays directory listing when no 'index.php' or index.html is present.
- `status`: displays server status/stats
- `env`: clearing/setting of ENV variables
- `negotiation`: content negotiation (e.g. different languages)
- `filter`: smart filtering of web requests
- `version`: handling version information in config files (<IfVersion>)
- `userdir`: - ~username in the URL will be mapped to the user's home directory
- `as-is`: - "as-is" filetypes.
- the various "authx_*" modules: (except the 'authz_host' module)
- `mod_vhost_alias`: the virtualhost support using header information

TIP: You don't need the 'mod_vhost' module to run multiple virtual host in Apache. This module is used to dynamically configure virtual hosts on the fly by using IP address and/or the 'host:' header of the HTTP request.)

ref:

http://httpd.apache.org/docs/2.2/mod/mod_vhost_alias.html

NOTE: You **do** need the '**mod_actions**' and '**mod_alias**' for Apache to work with PHP-FPM

3. Disable directory Indexing

The default settings allow someone to browse the content of a directory if there is no index file (e.g. index.php, index.html) found in that directory. This can be a serious security breach if there are sensitive documents/files in that folder which should not be accessed/viewed by the public. To prevent this, you can use the directive: -Indexes in the <Directory> container.

For example,

```
<Directory /var/www/html>
    Options -Indexes
</Directory>
```

Or, if you want to be more restrictive (or lock down) the top most webroot directory, use the directive 'Options None' instead. See the example in the next section (item #4) below.

4. Restrict Access to 'Root' directory

We can restrict access to various directories accessible by Apache by using the 'Allow' or 'Deny' directives in the <Directory> container. For example, the root directory container:

```
<Directory />
    Options None
    AllowOverride None
    Order allow,deny
    Allow from all
</Directory>
```

* **Options None** – this means no optional extra features are allowed.

The directive: **“Options”** has the following 'parameters/options'

- Options *All* - all options are enabled (except MultiViews). If you don't specify Options directive, this is the default value.
- Options *ExecCGI* - Execute CGI scripts
- Options *FollowSymLinks* - symbolic links in this directory will be followed.
- Options *SymLinksIfOwnerMatch* - Similar to FollowSymLinks, but it will only follow if the owner of the link and the original directory is the same.
- Options *Includes* - Enable server side includes (uses mod_include)
- Options *IncludesNOEXEC* - Enable server side includes without the ability to execute a command or cgi.
- Options *Indexes* - Enable directory listing
- Options *MultiViews* - Allow content negotiated multiviews (uses mod_negotiation)

The directives: “**AllowOverride**” has the following 'parameters/options'

- AllowOverride *All* -allows the individual virtual hosts to override the settings via the .htaccess file
- AllowOverride *None* - the opposite of “AllowOverride All” above.
- Allow *AuthConfig* - enables authorization directives
- AllowOverride *Indexes* - enable the automatic directory listing

- `AllowOverride Limit` - enable directives controlling host access (Allow,Deny,Order)
- `AllowOverride Options` - enable directives controlling specific directory features.
- `AllowOverride FileInfo` - enables directives controlling document types. For example, to enable .htaccess for url rewrites in Wordpress, you can use this directive.

For a complete list and more details, visit:

<http://httpd.apache.org/docs/2.2/mod/core.html#allowoverride>

Warning: using a restrictive <Directory> container such as “Options None”, and “AllowOverride” none above will most likely break your websites. For example, Wordpress which uses the .htaccess for its url rewrite (pretty permalinks) will be broken.

The idea is to set a restrictive document root, and relax the individual directories below the document root as required by the individual websites.

Here's an example on how to work around this restriction by inserting another 'AllowOverride' directive within the <Directory> container WITHIN the VirtualHost:

```
<VirtualHost X.X.X.X:80>
  ServerName ponyvps.com
  :
  <Directory /home/pony/www/ >
    AllowOverride FileInfo
  </Directory>
  :
</VirtualHost>
```

The '***AllowOverride FileInfo***' directive will enable the Wordpress' .htaccess file to work as normal, but only in the "/home/pony/www/" directory, but NOT in the root directory.

- Order allow,deny - This is the order in which the "Allow" and "Deny" directives should be processed. This processes the "allow" first and "deny" next.
- Deny from X.X.X.X - This denies all request from the IP address 'X.X.X.X'

Although you can ban certain IPs via the 'Deny' directive in Apache, it is more efficient and effective to do so via the Server Firewall.

5. Limit request Size

The default settings in Apache does not place a limit on the size of the HTTP request, i.e. in other words, someone who wants to bring down your website can easily do so by sending multiple large (e.g. 100MB) packet requests to Apache. This will cause a Denial-of-Service (DOS) condition. To prevent this, you can set the limit by using this directive:

"LimitRequestBody" which is placed within the <Directory> container.

Example:

```
<Directory />
    Options None
    AllowOverride None
    Order allow,deny
    Allow from all
    LimitRequestBody 1048576
</Directory>
```

*note: 1048576 bytes = 1MB (1024 * 1024)

This limits the maximum request size to just 1MB. You will need to adjust this accordingly to your websites/web-applications as some sites may need more than just 1MB.

Most PHP/Wordpress powered websites sets the limit between 8MB - 12MB

6. Symlink Vulnerability

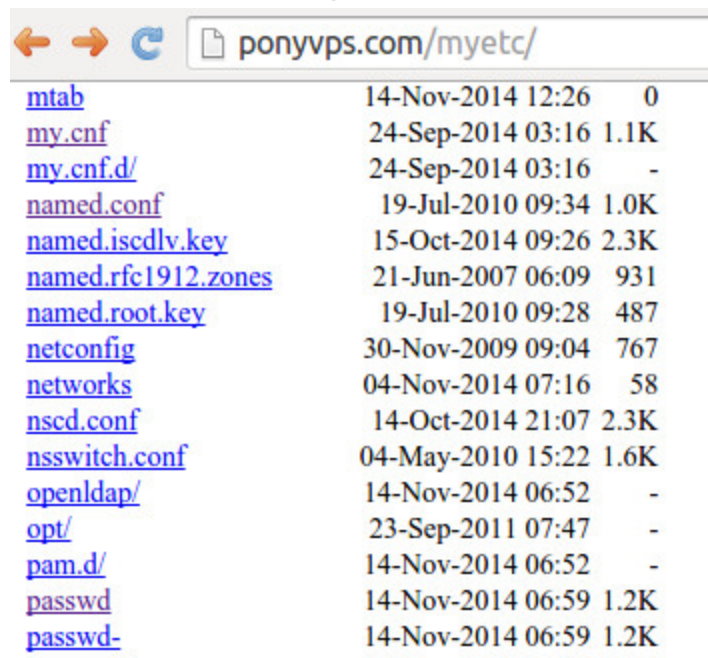
This directive, "FollowSymLinks" in the <Directory> container allows Apache to follow symbolic links (references) in the file systems. For example, the default (unsecured) httpd.conf file allows FollowSymLinks, i.e.

```
<Directory />
    Options FollowSymLinks
    AllowOverride None
    Order allow,deny
    Allow from all
</Directory>
```

This allows a user to create a symbolic (symlink) such as:

```
$ ln -s /etc myetc
```

This allows the user to point his browser to: <http://ponyvps.com/myetc/> to view all the files in the system /etc/ directory IF the 'Options Indexes' is enabled (the default), e.g:



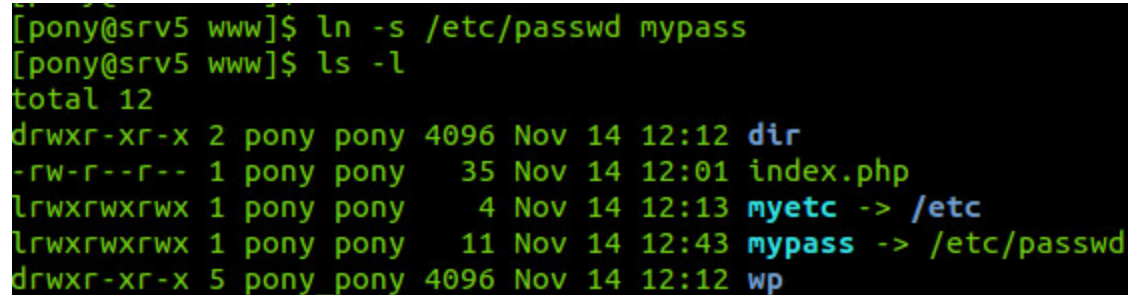
mtab	14-Nov-2014 12:26	0
my.cnf	24-Sep-2014 03:16	1.1K
my.cnf.d/	24-Sep-2014 03:16	-
named.conf	19-Jul-2010 09:34	1.0K
named.iscdlv.key	15-Oct-2014 09:26	2.3K
named.rfc1912.zones	21-Jun-2007 06:09	931
named.root.key	19-Jul-2010 09:28	487
netconfig	30-Nov-2009 09:04	767
networks	04-Nov-2014 07:16	58
nscd.conf	14-Oct-2014 21:07	2.3K
nsswitch.conf	04-May-2010 15:22	1.6K
openldap/	14-Nov-2014 06:52	-
opt/	23-Sep-2011 07:47	-
pam.d/	14-Nov-2014 06:52	-
passwd	14-Nov-2014 06:59	1.2K
passwd-	14-Nov-2014 06:59	1.2K

Clicking on any of the files will display the content if the file has 'read' permission for the group 'others'.

If the “Options Indexes” is turned off (e.g. Options - Indexes), then directory listing is disabled, but a user can still symlink directly to the file if the user knows the name, for example, the /etc/passwd file definitely exist in every Linux system and has 'read' permission (644) for the group others.

A user can then issue this command:

```
$ ln -s /etc/passwd mypass
```



```
[pony@srv5 www]$ ln -s /etc/passwd mypass
[pony@srv5 www]$ ls -l
total 12
drwxr-xr-x 2 pony pony 4096 Nov 14 12:12 dir
-rw-r--r-- 1 pony pony  35 Nov 14 12:01 index.php
lrwxrwxrwx 1 pony pony   4 Nov 14 12:13 myetc -> /etc
lrwxrwxrwx 1 pony pony  11 Nov 14 12:43 mypass -> /etc/passwd
drwxr-xr-x 5 pony pony 4096 Nov 14 12:12 wp
```

Now, when the user browse to <http://ponyvps.com/mypass> – he will be able to view the content of the system password file and see **ALL** the userIDs in this system.

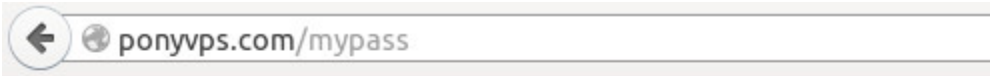
```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
gopher:x:13:30:gopher:/var/gopher:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:/:/sbin/nologin
vcsa:x:69:69:virtual console memory owner:/dev:/sbin/nologin
sasauthd:x:499:76:"Sasauthd user":/var/empty/sasauth:/sbin/nologin
mailnull:x:47:47:/:/var/spool/mqueue:/sbin/nologin
smmsp:x:51:51:/:/var/spool/mqueue:/sbin/nologin
tcpdump:x:72:72:/:/sbin/nologin
apache:x:48:48:Apache:/var/www:/sbin/nologin
named:x:25:25:Named:/var/named:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
rpc:x:32:32:Rpcbind Daemon:/var/cache/rpcbind:/sbin/nologin
nscd:x:28:28:NSCD Daemon:/:/sbin/nologin
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
pony:x:501:501:/:home/pony:/bin/bash
donkey:x:502:502:/:home/donkey:/sbin/nologin
```

Now, if this is a malicious user, he is able to grab all the userIDs in this system and see which user has shell access, for example, the user “pony” has shell access, but the user “donkey” does not (the shell is set to /sbin/nologin). So, this user can run password crackers against the user 'pony.

But if in the /etc/httpd/conf/httpd.conf file – you have this directive:

```
<Directory />
    Options None
    AllowOverride None
</Directory>
```

Then, the user will get this “Permission Denied” error:



Forbidden

You don't have permission to access /mypass on this server.

But take note, but doing this, many of the web applications/CMS such as Wordpress, Drupal, Joomla, etc – will break unless you enable them again within the Virtualhost container, e.g.

```
<VirtualHost X.X.X.X:80>
  ServerName ponyvps.com
  :
  <Directory /home/pony/www/ >
Options All
    AllowOverride FileInfo
  </Directory>
  :
</VirtualHost>
```

7. TraceEnable Off

The 'HTTP TRACE' request is used as a debugging tool to echo all the received information. A skilled hacker can use this feature (if enabled) to print HTTP cookies and steal HTTP sessions to view sensitive data. This request is usually used in the XSS (Cross Site Scripting Attack). You should always disable this on production servers.

Note: This directive is usually not shown in the 'httpd.conf' file, so, just insert it near the top, with the value 'Off', i.e.

TraceEnable Off

8. Freeze the Apache's config file:

Once you're happy with the changes to the Apache configuration file, you may want to prevent anyone from accidentally changing it. You can do this by setting the

immutable bits to the file by running the following commands:

```
# chattr +i /etc/httpd/conf/httpd.conf
```

This will prevent yourself (root) from accidentally deleting/modifying the file without first removing the immutable bits by running the following commands:

```
# chattr -i /etc/httpd/conf/httpd.conf
```

9. Use 'mod_security'

This is advance stuff. There are volumes and books written on 'mod_security' and its syntax is quite arcane. Sufficient to say, it's not for the faint-hearted. We won't be covering this topic in this book, but if you are interested, here's the official web site for more info:

<https://www.modsecurity.org/>

Essentially, the 'mod_security' is an application firewall – e.g. a 'personal firewall' built into Apache itself as a module. Every web requests to Apache will be inspected by 'mod_security' for malicious requests/data.

Note that using this module will have a performance impact since every web-requests to Apache will be inspected by this module for malicious URIs.

Chapter 6. Securing PHP (php.ini)

In the previous chapter, we have seen how the default 'Options FollowSymLinks' directive in the Apache httpd.conf file enables any user to symlink to any system file and view them if the file permissions has the 'read' access to the group 'others'.

If you absolutely must use the FollowSymLinks directive, then use:

Options SymLinksIfOwnerMatch

This tells Apache to follow the symlink ONLY if the target file belongs to the same userID. In this scenario, a user who symlinked to the /etc/passwd file cannot view its content because the /etc/passwd file belongs to root and not the user.

In this chapter, we'll see how although we have secure the Apache webserver, it is still possible to use PHP to run system commands to list files, delete files, etc. to bypass the security settings we have set up in Apache.

Because of this, it is not advisable to leave the default "php.ini" as is because the default settings are very permissive and allows a user to run almost any command supported by PHP such as various system calls.

For example, if a user cannot view the content of the /etc/passwd file because we have disabled the followsymlink directive, the user can still cobble a simple 1 liner PHP script to view the content of that file, e.g.

```
<?php echo '<pre>'; system("cat /etc/passwd"); echo '</pre>'; ?>
```

That simple 1-liner will display the entire content of the password file! Take a look:


```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:/:/sbin/nologin
vcsa:x:69:69:virtual console memory owner:/dev:/sbin/nologin
saslauthd:x:499:76:"Saslauthd user":/var/empty/saslauth:/sbin/nologin
mailnull:x:47:47:/:/var/spool/mqueue:/sbin/nologin
smmsp:x:51:51:/:/var/spool/mqueue:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
apache:x:48:48:Apache:/var/www:/sbin/nologin
named:x:25:25:Named:/var/named:/sbin/nologin
rpc:x:32:32:Rpcbind Daemon:/var/cache/rpcbind:/sbin/nologin
tcpdump:x:72:72:/:/sbin/nologin
nscd:x:28:28:NSCD Daemon:/:/sbin/nologin
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
donkey:x:500:500:/:/home/donkey:/sbin/nologin
pony:x:501:501:/:/home/pony:/bin/bash
```

NOTE: This is the **DEFAULT** settings in PHP in 90% of all the web-servers!

To prevent this from happening, we need to lock down our PHP settings by editing the main configuration file - /etc/php.ini

1. Disable System Functions

The most important step is to disable all the 'dangerous' system functions. Open the php.ini file and look for:

disable_functions =

The default value is empty, meaning, PHP can run any commands that a normal system user can. Replace that line with:

```
disable_functions = "symlink, dl, exec, shell_exec, system, passthru, popen, pclose, proc_open, proc_nice, proc_terminate, proc_get_status, proc_close, pfsockopen, leak, apache_child_terminate, posix_kill, posix_mkfifo, posix_setpgid, posix_setsid, posix_setuid, show_source, allow_url_fopen"
```

Save the php.ini file and restart php-fpm. Then browse: <http://ponyvps.com/showpass.php> again – you should see a 'blank page this time because we have disabled the 'system' command. But a crafty user can still bypass that using the common “file()” function.

We could add the file() function to the list of disable_functions list, but it will probably cause some existing/legacy web applications to break because the file() function is commonly used. The other alternative is to restrict the location where PHP is allowed to access files by tweaking the 'open_basedir' directive.

2. Restrict the “open_basedir”

Earlier, we had used a 1 liner: `<?php system("cat /etc/passwd"); ?>` to display the content of the passwd file. We stopped that by editing the 'disable_functions' directive in the php.ini file. But there's another way to view the content of the passwd. Change the code in showcode.php to:

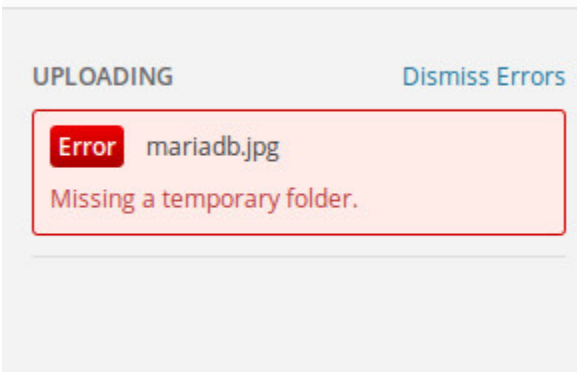
```
<?php $dat = file("/etc/passwd");  
foreach ($dat as $x ) { echo "$x<br />"; }  
?>
```

and point your browser to the page again. You will see the content of the /etc/passwd file. This is because PHP has access to the entire file system. To prevent this, we need to restrict PHP to access files in the DocumentRoot (and its subdirectories only) via:

```
open_basedir = /home  
(or whichever path you've set your DocRoot to be, e.g.  
open_basedir = /var/www/html )
```

Now, if you try to view: <http://ponyvps.com/showpass.php> – you will see a blank page again.

WARNING: if you set the “open_basedir = /home” - it will break some applications that needs to temporarily write access to the /tmp directory - e.g. when uploading media files (images) in your WordPress dashboard. For example, when I tried to upload an image in Wordpress dashboard, it complained by saying it can't find a temporary folder.



So, in order to work around it, we need to create a new 'tmp' directory for PHP to use. It should be under the "/home" directory since we have restrict PHP to /home.

```
# mkdir /home/tmp
# mount -t tmpfs -o nodev,noexec,nosuid tmpfs /home/tmp
# chmod 1777 /home/tmp
# echo "none /home/tmp tmpfs nodev,noexec,nosuid 0 0" >> /etc/fstab
```

Note the last command has ">>" and NOT ">". If you use ">" - you would have clobbered the /etc/fstab file. If you're unsure, manually edit the /etc/fstab to add this new entry.

Warning!

The /etc/fstab is a critical system file - if it's corrupted, then your server may fail to start-up when it's rebooted. So, always have a backup copy in place. Edit this file with caution.

The content of your `/etc/fstab` will most likely to be different from mine, especially if yours is a dedicated server or another Linux distro.

Ok, with that out of the way, we need to edit the `php.ini` file and specify a different directory for file-uploads:

```
upload_tmp_dir = /home/tmp
```

WARNING!:

Now that we have restricted PHP to the `/home` directory, and a user cannot use a PHP script to read/view contents of files outside `/home` (e.g. viewing `/etc/passwd` will fail). However, this does not prevent a user from writing a PHP script to read files from another USER's directory if the permission settings these user directories are set to `"755"` or `"711"`.

For example, if a hacker knows the user 'pony' is running Wordpress, and the hacker knows the 'home' directory of that user (e.g. by viewing the contents of `/etc/passwd`), then the hacker can easily modify the script to read the `'wp-config.php'` file:

```
<?php
$dat = file("/home/pony/public_html/wp-config.php");
foreach ($dat as $x ) { echo "$x<br />"; }
?>
```

And if the directory `"/home/pony"` is set to `"755"` or `"711"`, then this user will be able to see the entire content of the `wp-config.php` file!

To prevent this, the directory permission should be set to `"750"` but this will only work IF:

- PHP-FPM process pool is running under its own userID, e.g. `"pony"`
- The listen mode is set to: `"listen.mode=0660"`

- The listen group is set to: “listen.group=pony”
- AND the Apache userID (“apache”) is added to the Secondary Group “pony”

Restart php-fpm and you’re good to go. (`# service php-fpm restart`)

3. Disable Remote/URL inclusion

You can even allow a PHP script to 'include' a remote script and execute its code. There are 2 types of remote inclusion directives:

allow_url_fopen = Off

- The directive tells PHP whether to allow the treatment of URLs (e.g. `http://` or `ftp://`) as files. Enabling this directive allows you to use PHP to retrieve and parse (external) webpages as if it's a local file .

allow_url_include = Off

- The directive above tells PHP whether to enable the functions 'include', 'include_once', 'require', 'require_once' to include remote scripts. This can be very dangerous if the output of the remote script is malicious php codes.

If none of your existing websites or web-applications need them, it should be set to “Off”.

Below are some of the commonly used directives and their explanation:

display_errors = Off

display_startup_errors = Off

- *On production servers, it is not advisable to display errors on the webpage because it may give clues to hackers on how PHP is configured*

html_errors = Off

- *When PHP encounters errors, it has the capability to insert html links to the documentation related to that error. For performance and security reasons, it is recommended that this directive is disabled on production servers*

short_open_tag = Off

- *this directive enables PHP to recognize code between <? and ?> tags as PHP source code. But with the wide spread use of XML and the use of these tags by other languages, the server may be confused and such, may parse codes incorrectly. Default value is 'On'. Recommended: 'Off'*

register_globals = Off

- *Ideal scenario is to turn this off because using form variables as globals can lead to possible security issues if the code is not well thought of (ref: <http://php.net/register-globals>)*

magic_quotes_gpc = Off

- *'Magic Quotes' is a preprocessing function which attempts to escape any character sequences in GET, POST, COOKIE and ENV data which might otherwise corrupt data being inserted into databases. However, due to different character encodings and*

non-standard SQL implementation across the different database engines, the 'magic_quotes' is not 100% accurate. This directive is deprecated as of PHP 5.3.0 and will be removed in PHP 6. It is better to handle the pre-processing of data in the application code itself. Recommended setting is 'Off

error_reporting: E_ALL & ~E_DEPRECATED

- *This is the recommended settings on production servers. The 'E_ALL' = all errors and warnings. E_DEPRECATED warns about code that may not work in future versions of PHP*

session.use_only_cookies = 1

- *This directive forces PHP to fetch and use cookie for storing and maintaining the session id. This is useful in combatting session hijacking when an application does not specify or manage its own session IDs. It is not fool-proof, but a good start to session hijacking defense*

Note: The other security settings are the 'safe_mode' directives which has been deprecated as of PHP 5.3 and is removed as of PHP 5.4. Basically, it is a quick hack by the PHP developers to mimic the UID/GID permissions of the unix system users. When it is enforced, PHP will check that the files/directories that a script is trying to access belonged to the user that invoked the script. Here's a good article explaining why this 'safe_mode' does not provide real security:

ref:

http://ilia.ws/archives/18_PHPs_safe_mode_or_how_not_to_implement_security.html

The following directives in the `/etc/php.ini` file are performance related. We will cover them now as the next chapter deals with Performance tuning in Apache.

Performance Directives in PHP

max_input_time = 60

- *Max amount of time a script may spend parsing request data. The default: -1, which means unlimited time. On production servers, it should not be longer than 120 seconds to prevent an abusive script from tying up server resources*

max_execution_time = 30

- *Max execution time for each script in seconds. Depending on your website and applications, you may need to increase it to 60 or 90*

memory_limit = 128M

- *Max amount of memory a script may use. This value will depend on how much memory/RAM your server has and also the applications on your website.*

register_argc_argv = Off

- *The default is 'On' - which tells PHP to register the `$argv` and `$argc` variables each time it runs. `$argv` contains an array of all the arguments passed to PHP when a script is invoked. `$argc` contains the number of arguments that were passed when the script was invoked. These arrays are useful when running scripts from command line (CLI), but not as much when running PHP on the web.*

- *Enabling this directive consumes CPU cycles and memory each time a script is invoked. For best performance, it should be disabled on production servers*

auto_globals_jit = On

- *If enabled, the SERVER and ENV variables will be created only when they're first used (Just In Time) instead of when the script is invoked. To use this directive, the following directives must be turned Off: register_globals, register_long_arrays and register_argc_argv.*
- *Using this directive instead of the individual register_* will result in performance gain if these variables are not required in the PHP scripts.*

register_long_arrays = Off

- *This directive determines whether the deprecated long \$HTTP_*_VARS type of predefined variables are registered with PHP or not. It is enabled by default for backward compatibility*

realpath_cache_size = 16k

- *This directive sets the size of the realpath_cache used by PHP for file operations. If your website/application opens many files, consider increasing the default value of 16K to boost performance*

realpath_cache_ttl = 120

- *This is the duration of time (seconds) for which to cache the realpath information for a given file or directory. If the files on your webserver rarely*

change, consider increasing the value (in seconds) for better performance

define_syslog_variables = Off

- *Turning this directive Off will increase performance*

mysqlnd.collect_statistics = Off

- *This directive enables/disable the collection of general statistics by mysqlnd which can be used to monitor MySQL operations. On production servers, turn this Off because you should use MySQL's own directives (e.g. enable slow_query_log) to monitor performance*

session.gc_probability = 1 & session.gc_divisor = 100

- *These 2 variables define the probability of running the PHP garbage collection process on every session initialization. The probability is calculated via $gc_probability / gc_divisor$ - so, a value of 1/100 will give you approximately 1% chance that GC (garbage collection) will run on any given request. On high traffic web servers, consider setting the `session.gc_divisor = 1000` because running the GC is CPU intensive.*
- *Note: PHP has an internal 'root buffer' which holds all the resources that PHP is currently using (tmp data, variables, pointers, etc) - once this buffer is full (i.e. 10,000 objects), PHP will automatically trigger the GC routine to clean-up.*
- *The "session.gc_*" directives here specifies the probability a user/web-request triggering the GC routine instead of the automated GC routine when the 'root buffer' is full*

Chapter 7. Performance Tuning

In the last section of the previous chapter, we've seen some of the important directives in the "php.ini" file which will boost the PHP-FPM performance. In this chapter, we'll look at optimizing Apache and tuning the PHP-FPM pool configuration for maximum performance.

This chapter will have 4 sections:

7.1 Memory Constraints

7.2 Tuning Apache Server

7.3 Tuning PHP-FPM pool configuration

7.4 Small Case Study

7.1 Memory Constraints.

The main principle in Server Performance tuning is to ensure that your server does not begin swapping under heavy load/traffic. This is the ***NUMBER #1 performance killer***.

Everything else is secondary. Once your server begins to swap heavily because it ran out of memory, performance will degrade rapidly because the Operating System will have to swap processes from RAM to disk and vice-versa.

So, the first thing we need to do is determine how much available memory you have on your server. If your server is a dedicated Apache/PHP-FPM server, then you can afford to be more generous with the memory allocation. But if your server is running many multiple services such as MySQL, Bind/NSD, Sendmail/Exim, FTPD, etc - then you have to consider memory allocation to those services as well.

To get an overview of how much memory your server has, and how much is available:

```
# free -m
```

```
[root@srv5 ~]# free -m
              total        used        free      shared    buffers     cached
Mem:           1024          374          649          140           0          152
-/+ buffers/cache:          222          801
Swap:           512           0          512
[root@srv5 ~]#
```

From the screenshot above, the amount of available memory is 801 MB. This value is taken from the 'buffers/cache' row (the 2nd row) - under the 'free' column.

This figure is calculated from the columns in the 1st row: 'free' + 'buffers' + 'cache'

Available memory: 'free' (649) + 'buffers' (0) + 'cached' (152) = 801 MB

The values in the 'buffers' and 'cached' (from the first line) are available for use by various processes when the need arises. The Linux OS uses the cache and buffers whenever it can to optimize server performance - but when a process needs extra memory, the 'buffers' and 'cached' will be released and re-allocated to those processes.

For more information, please visit:
<http://www.linuxatemyram.com>

Note: Ideally, you should take this reading during the server's normal/peak hours so that all the other services are running and using memory.

The next step is to find out how much memory each Apache process and PHP-FPM is using:

Use the quick 1-liner command to calculate:

Apache Processes:

```
# ps -ylC httpd | awk '{x += $8;y += 1} END {print "Apache Mem Usage (MB): "x/1024; print "Avg Process Size (MB): "x/((y-1)*1024)}'
```

PHP-FPM Processes:

```
# ps -ylC php-fpm | awk '{x += $8;y += 1} END {print "PHP-FPM Mem Usage (MB): "x/1024; print "Avg Proccess Size (MB): "x/((y-1)*1024}}'
```

Once you have the numbers, you're able to correctly set the "max" number for the Apache and PHP-FPM child-processes. For example:

```
[root@srv5 ~]# ps -ylC httpd | awk '{x += $8;y += 1} END {print "Apache Mem Usage (MB): "x/1024; print "Avg Proccess Size (MB): "x/((y-1)*1024}}'
Apache Mem Usage (MB): 7.91797
Avg Proccess Size (MB): 2.63932
[root@srv5 ~]#
[root@srv5 ~]# ps -ylC php-fpm | awk '{x += $8;y += 1} END {print "PHP-FPM Mem Usage (MB): "x/1024; print "Avg Proccess Size (MB): "x/((y-1)*1024}}'
PHP-FPM Mem Usage (MB): 179.441
Avg Proccess Size (MB): 9.96897
[root@srv5 ~]#
```

On my system, each Apache child process is using 2.6MB of memory and each PHP-FPM process is using 10MB.

Assuming that I have 700MB memory to allocate to Apache and PHP-FPM, and that I have only 2 virtualhost (sites) on my server, i.e. PonyVPS.com and DonkeyVPS.com – I would set my config to:

httpd.conf

```
<IfModule worker.c>
StartServers      1
MaxClients        100
:
</IfModule>
```

/etc/php-fpm.d/pony.conf

```
pm.max_children = 20 (max processes)
pm.start_servers = 5
```

/etc/php-fpm.d/donkey.conf

```
pm.max_children = 20 (max processes)
pm.start_servers = 5
```

Max Apache memory Usage: $100 \text{ (max clients)} * 2.6\text{MB} = 260 \text{ MB}$

Max PHP-FPM memory Usage: $20 * 10\text{MB} * 2 \text{ (sites)} = 400\text{MB}$

Total memory usage = 660MB (which is below our 700MB limit).

If I have 4 virtual hosts (instead of 2), then I will need to scale down the “pm.max_children” Your situation will be different of course, but if you follow this “rule-of-thumb” - i.e. do not set the configuration settings more than the available memory, you will prevent your server from swapping.

(Note: In a live environment, I would actually reduce those numbers by 20% to be on the safe side, and I would monitor the server logs for 'Out-Of-Memory' (OOM) messages)

7.2 Tuning Apache Server

In this section, we will tweak a few important directives in the ‘httpd.conf’ and unload the un-necessary modules to make Apache as lean (small) as possible. We will also look at the impact of having the .htaccess file on performance. The table below describes the various directives and their implications:

Apache Performance Directives:

Timeout 60

- *The number of seconds before a web request timed out. Set this value to between 60 - 100 seconds to prevent small denial of service (DOS) attacks by script kiddies from overwhelming your server*

DirectoryIndex index.php index.html

- *Tells Apache specifically what are the “index” files instead of making Apache ‘guess’ (e.g. when using “DirectoryIndex index”)*

HostnameLookup Off

- *This will prevent Apache from doing reverse-name lookup from the IP addresses of the website visitors. This is a great performance booster because doing a name-server lookup is time consuming*

ExtendedStatus Off

- *If mod_status is enabled, this will prevent Apache from issuing several extra time-related system calls on every requests made*

KeepAlive On/Off

- *This tells Apache whether to keep an existing/open TCP connection “alive” after the request has been fulfilled. This allows the same client to send multiple requests over the same connection and thus bypass the overhead of setting up new TCP connections)*
- *Setting KeepAlive On is resource intensive, although in some cases, this has shown to boost performance considerably when using the 'worker' MPM.*
- *In the case of 'prefork' MPM, performance suffers because a client can tie-up an entire process for that span of time, even if that request was for a static image file. This is not an issue with the 'worker MPM' which is thread-based.*
- *If you are using Nginx as the reverse-proxy, then set it to “Off”.
Otherwise, keep it “On” and set the KeepAliveTimeout small*

KeepAliveTimeout 3

- *This is the number of secs Apache will keep a TCP connection alive while waiting for the same client to send the next request.*
- *If the client did not send another request within this time frame, that connection will be dropped. This will prevent Apache from waiting for an idle client. Keep it small (3-5 sec)*

MaxKeepAliveRequests 250

- *This is the number of requests allowed during a persistent TCP connection. Keep this high (200 - 500) so that Apache can serve a busy client using the same connection*

.htaccess

One of the often overlooked performance booster is the usage of .htaccess file. If possible, do not use the .htaccess file as it slows down Apache's performance. When .htaccess is enabled, Apache will search/look for this file in every directory & subdirectory in the document root.

This will cause serious performance hit because it requires reading from the disk, especially if there are tons of directories and subdirectories. All the commands/directives that can go into the .htaccess file, you can put them into the '<Directory>' container in the main Apache configuration file (httpd.conf).

For example, if you have a dedicated server (or VPS), and you're using it for your own websites, then you can easily do this and not use .htaccess to boost Apache's performance.

Here's an example of Wordpress 'pretty permalinks' directive in the .htaccess file:

```
# BEGIN WordPress
<IfModule mod_rewrite.c>
RewriteEngine On
RewriteBase /
RewriteRule ^index\.php$ - [L]
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule . /index.php [L]
</IfModule>
```

We can embed the same code above into the <Directory> container in the httpd.conf file.

```
<VirtualHost *:80>
    ServerName donkeyvps.com
    :
    # BEGIN WordPress
    <IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /
    RewriteRule ^index\.php$ - [L]
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteRule . /index.php [L]
    </IfModule>
    # END WordPress
    :
</VirtualHost>
```

This will do exactly the same thing as having the .htaccess in the ponyvps.com document root, except that performance will be much faster!

But if you're providing hosting services to the public, then you'll have to enable .htaccess so that your clients can manage the directives in .htaccess themselves.

Keep the Apache Modules to a bare minimum

This will make Apache less 'bloated' and consume less memory per child-server. The savings in memory will enable you to increase the number of MaxClients directive.

Depending on how many modules are loaded, each apache process can use anywhere between 1.5MB up to 16MB or even more. This will cause a lot of strain on the server and makes Apache sluggish. One module to disable is the: ***“mod_mime_magic”***

This particular module' can cause serious performance hit for busy sites because it causes Apache to look at the first few bytes of the content of the file requested to guess the content type and encoding.

Unless your site has some esoteric type of files, disable “mod_mime_magic”. Specify the mime types via the “TypesConfig” directive, i.e. *TypesConfig /etc/mime.types*

The file (/etc/mime.types) contains the information on various different file types that Apache recognizes. Another module which you should consider removing is the: ***“mod_negotiation”***

This module enables Apache to choose the best representation of a resource based on the browser-supplied preferences for media type, languages, character set and encoding. It also implements a couple of features to give more intelligent handling of requests from browsers that send incomplete negotiation information.

For example, a browser indicates that it would like to see information in French if possible, but if not, then the English language. (Browsers indicate their preferences by headers in the request). This module will try to accommodate the browser's request.

Depending on your main target audience, you may or may not want to enable mod_negotiation (disabling this module

will increase performance), e.g. if your website content is mainly for the English speaking audience, then disable this module.

These 2 modules should be enabled if possible to enhance performance:

- **mod_deflate**

This module enables Apache to compress HTML, text, Javascript, CSS files to around 25% of their original sizes before sending it to the web-browser. This saves on bandwidth and speed things up.

- **mod_expires**

Include mod_expires for the ability to set expiration dates for specific content; utilizing the 'If-Modified-Since' header cache control sent by the user's browser/proxy. Will save bandwidth and drastically speed up your site for [repeat] visitors.

To view what are the enabled modules in Apache:

```
# apachectl -M
```

To count the number of loaded modules:

```
# apachectl -M | wc -l
```

The default Apache setup has around 50+ modules enabled. You definitely do not need all these modules. Here is a list of modules required for Apache to work with PHP-FPM and run popular CMS such as Wordpress.

```
# apachectl -M
```

```
core_module (static)
mpm_worker_module (static)
http_module (static)
so_module (static)
authz_host_module (shared)
```

```
include_module (shared)
log_config_module (shared)
expires_module (shared)
deflate_module (shared)
headers_module (shared)
setenvif_module (shared)
mime_module (shared)
actions_module (shared)
alias_module (shared)
rewrite_module (shared)
dir_module (shared)
fastcgi_module (shared)
```

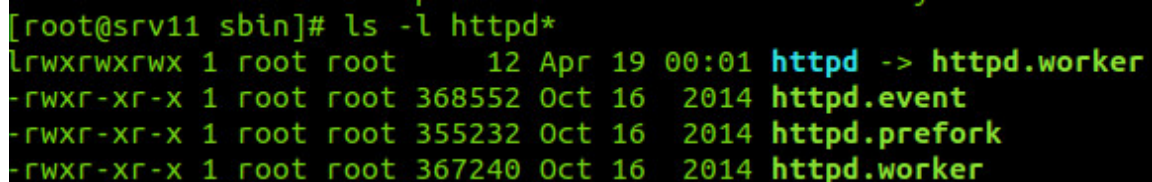
Choice of Apache's MPM

The Apache's core engine is known as the 'MPM' (Multi-Processing-Module). The 2 well known MPMs are the 'prefork' and the 'worker'. The 'prefork' MPM is the default choice by shipped by CentOS, Ubuntu and many other Linux distros because it is the most stable and works with many different web applications.

If you are running Apache + PHP-FPM as outlined in this book, you will get better performance if you select the "worker" MPM instead of the default "prefork" MPM.

To do this, you just need to symlink to the correct Apache binary in the /usr/sbin/ directory:

```
# cd /usr/sbin && ls -l httpd*
# cp httpd httpd.prefork && rm httpd
# ln -s httpd.worker httpd && ls -l httpd*
# service httpd restart
```



```
[root@srv11 sbin]# ls -l httpd*
lrwxrwxrwx 1 root root    12 Apr 19 00:01 httpd -> httpd.worker
-rwxr-xr-x 1 root root 368552 Oct 16 2014 httpd.event
-rwxr-xr-x 1 root root 355232 Oct 16 2014 httpd.prefork
-rwxr-xr-x 1 root root 367240 Oct 16 2014 httpd.worker
```

NOTE: If you have many "legacy" web-applications (that are not thread-safe), then choosing the 'prefork' MPM is a safer bet. Otherwise, opt for the "worker" MPM as it is more resource-friendly and provides better performance.

The table below lists the differences between these 2 MPMs:

MPM: <u>prefork</u>	MPM: <u>worker</u>
Multi-processes; 1 thread per process	Multi-processes; multiple threads per process
Each request/connection is handled by a process	Each request/connection is handled by a thread
This MPM is chosen mainly for stability and security.	This MPM is chosen for better performance and lower memory consumption.

<p>Summary: This MPM is used on sites that need compatibility with non-thread-safe libraries.</p> <p>Because each request is handled by a different process, it provides the best isolation – so that a problem with a single request will not affect any other.</p> <p>This MPM is very self-regulating, so it is rarely necessary to adjust its configuration directives.</p> <p>Most important is that <u>MaxClients</u> – it should be big enough to handle as many simultaneous requests as you expect to receive, but small enough to assure that there is enough physical RAM for all processes.</p>	<p>Summary: This is a hybrid multi-process, multi-threaded server and uses threads to serve requests.</p> <p>Because of this, it is able to serve a large number of requests with fewer system resources compared to '<u>prefork</u>' MPM.</p> <p>Note: it retains much of the stability of the '<u>prefork</u>' MPM by keeping multiple processes available, each with many threads.</p> <p>The important directives used to control this MPM are <u>ThreadsPerChild</u>, which controls the number of threads deployed by each child process and <u>MaxClients</u>, which controls the maximum total number of threads that may be launched.</p>
--	--

7.3 Tuning PHP-FPM

As we will see later in a small case-study/benchmark test, bigger numbers do not necessarily mean better performance. Every process and thread consumes memory and we've seen an example on how to calculate some numbers in Section 7.1

The default default values provided by CentOS and Ubuntu are conservative as well, e.g.

`pm.min_spare_servers = 5` (CentOS)

`pm.min_spare_servers = 2` (Ubuntu)

These conservative values instructs the PHP-FPM process manager keeps only 2 or 5 idle child-processes alive to accept connections. I've seen some folks setting this directive to 20 or 40 on a small VPS with just 1GB RAM.

In our example calculation in Section 7.1, having 40 spare/idle processes running means 400MB ($40 * 10\text{MB} = 400\text{ MB}$) of 'wasted' memory – which could be used by other services on the server. This is a sure recipe for disaster, especially if the site has heavy traffic.

Start with these values first and test it with the “**ab**” tool while monitoring the php-fpm error log, e.g.

```
# tail -f /var/log/php-fpm/error.log
```

If you see errors or warnings like the ones below, then you can increase the values slowly and re-test.

[22-October -2014 07:32:45] WARNING: [pool donkey] seems busy (you may need to increase pm.start_servers, or pm.min/max_spare_servers), spawning 32 children, there are 0 idle, and 15 total children

1. Using Unix Socket instead of TCP port

If possible, use the unix socket instead of the TCP port as it offers better performance – less context switches and copying of data and no setting up of TCP connection overhead).

But take note that if you configure PHP-FPM to listen to the local sockets, then PHP is only available to the other software (e.g. Apache or Nginx) running on the same server..

2. Calculating the 'pm.max_children' value:

In Section 7.1 we had used a one-liner command to extract the avg memory of the PHP-FPM process. It was about 10MB (in idle state).

However, in a live environment where your server is busy serving multiple client requests, the memory usage will be much higher. For example, a typical Wordpress site with multiple plugins may consume between 20-40MB per page request depending on how many plugins are active.

Assuming a typical php/wordpress page consumes 25MB, and we allocate 400MB for PHP-FPM, then the value for 'pm.max_children' would be: (available memory / process memory), i.e.

`pm.max_children = 400/25 = 16`

But since we have 2 sites (Pony & Donkey), and we need to halve that value again so that both sites will receive equal resources, i.e.

`pm.max_children = 8`

Note that this is just a base number to start off with. In the real environment, the Pony site could be very busy and receive lots of visitors, and the Donkey site hardly any visitors. As such, you can increase the pm.max_children setting in "pony.conf" and reduce it in the "donkey.conf".

As you monitor your sites over time, you are better able to adjust the settings accordingly for max performance.

3. Prevent Memory Leak

In an ideal situation, all the applications will release the server resources (eg. Memory, pointers, etc) once they are no longer needed. However, the reality is that there are many poorly coded apps, especially the many thousand plugins for Wordpress, Drupal, and Joomla and other CMSes.

This can cause some serious memory leak if a child-process is allowed to live “forever”.

This directive: “pm.max_requests” ensures that a child process is killed and respawned after it has processed the number of requests specifield here.

```
pm.max_requests = 500
```

This tells PHP-FPM to kill a child-process after it has served 500 requests to prevent memory leak. The table below describes some of the other directives for the PHP-FPM process pools.

Directives	Description
<pre>emergency_restart_threshold 10 emergency_restart_interval 1m process_control_timeout 10s</pre>	<p>The 'emergency_restart_threshold' and 'restart_interval' directives tells the PHP-FPM that it should restart itself automatically if there are 10 child processes that terminates with the SIGSEGV or SIGBUS signal within 1 minute.</p> <p>The 'process_control_timeout' sets a 10 seconds time limit for a child process to wait for signals from the master or process manager (pm).</p>
<pre>catch_workers_output = no</pre>	<p>This directive will cause PHP-FPM to log errors into the main error log. Default is “no”. In development environment, you can set this to ‘yes’, but remember to set it to “no” when going live.</p>

Here's a sample configuration on my server: /etc/php-fpm.conf

```
[global]
```



```

pid = /var/run/php-fpm/php-fpm.pid
error_log = /var/log/php-fpm/error.log
emergency_restart_threshold = 10
emergency_restart_interval = 1m
process_control_timeout = 10s
daemonize = yes
;; pool definitions ;;
include=/etc/php-fpm.d/*.conf

```

7.4 Small Case Study using Apache Bench (ab)

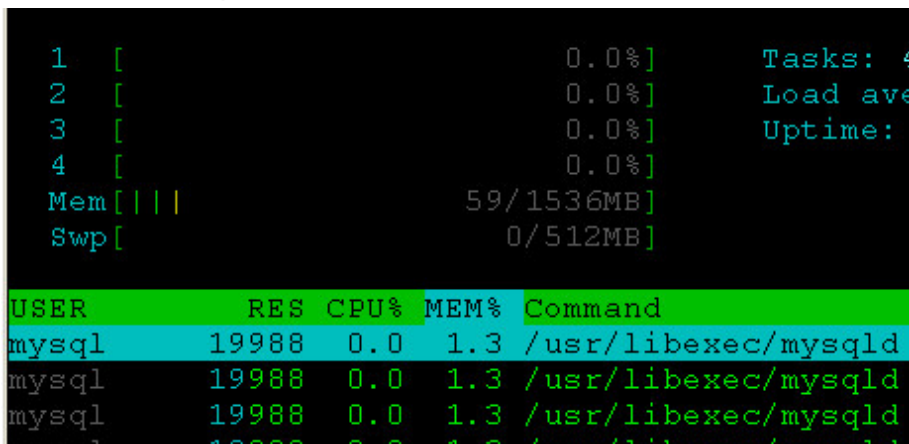
In this section, we will stress-test our setup with the “ab” software tool to simulate multiple concurrent web requests to our website. You may want to install the 'htop' software if it's not already installed on your server.

```
# yum install htop
```

We will use ‘htop’ to check monitor server status.

```
# htop
```

As you can see, the memory usage is only around 59MB when the system is idle.



The screenshot shows the htop interface. At the top, it displays system statistics: 1 task, 0.0% CPU, 0.0% load average, 0.0% uptime, and 59/1536MB memory usage. Below this, a table lists running processes. The MySQL process is highlighted in blue.

USER	RES	CPU%	MEM%	Command
mysql	19988	0.0	1.3	/usr/libexec/mysqld
mysql	19988	0.0	1.3	/usr/libexec/mysqld
mysql	19988	0.0	1.3	/usr/libexec/mysqld

We need to simulate real traffic to our server – preferably a “real” webpage that connects to MySQL. For example, setup Wordpress and create a few sample posts with a few images. Then use the “ab” tool to simulate requests to that page, e.g.

```
# ab -n300 -c5 http://donkeyvps.com/hello-world
```

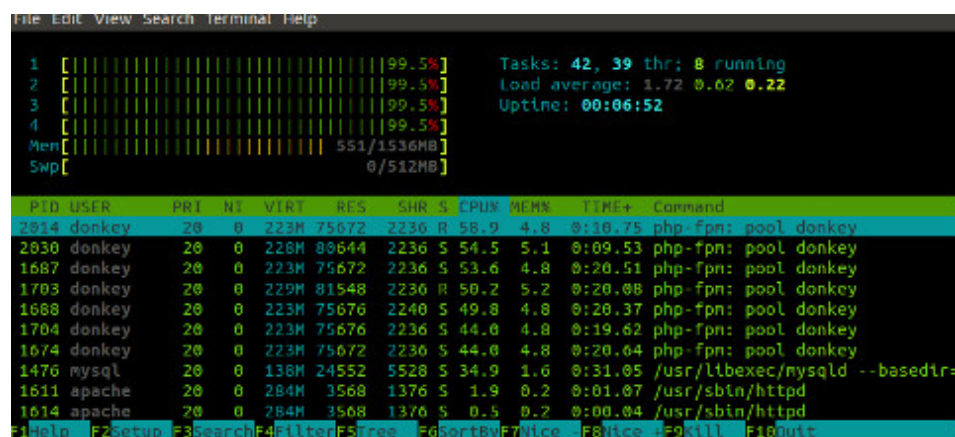
The command above sends 300 (5 concurrent) requests to the page:

Note:

-n is the number of requests to send, in our example above, 300

-c is the 'concurrent' level, i.e. in our case, send 5 concurrent requests, 300 times.

See an example screenshot below:



As the screenshot above, our 4-core CPU is nearly maxed (at 99%) and the memory used is only about 550MB - so, although we still have spare memory, we are limited by the CPU in this particular case.

Let's do 2 Simple Test Cases

1. Sending multiple concurrent requests to a CPU intensive script (calc.php)
2. Sending multiple concurrent requests to a typical Wordpress page with a few images.

Test Case (A) - CPU intensive script

Below are the 2 configurations for the PHP-FPM [donkey] pool:

Configuration #1:	Configuration #2:
<u>pm.max_children</u> = 50 <u>pm.start_servers</u> = 3 <u>pm.min_spare_servers</u> = 3 <u>pm.max_spare_servers</u> = 5 <u>pm.max_requests</u> = 500	<u>pm.max_children</u> = 100 <u>pm.start_servers</u> = 10 <u>pm.min_spare_servers</u> = 10 <u>pm.max_spare_servers</u> = 10 <u>pm.max_requests</u> = 10000

The command used to test: # ab -n300 -c5
<http://donkeyvps.com/calc.php>

Test Results:

Configuration 1	Configuration 2
Memory usage: 550 – 620 MB	Memory usage: 900MB – 1024 MB
Time taken: 20.68 seconds	Time taken: 25.57 seconds.
<u>Num of request/sec</u> : 14.51	<u>Num of request/sec</u> : 11.73

Observation: *Config #2* has more ‘resources’ allocated to the PHP-FPM, but its performance is poorer compared to *Config #1*. This is because when we allocate more ‘resources’ than required, it has to spend “extra” resources to manage them (e.g. the max_children, spare_servers, etc).

Max memory (config #2) during the test is around 1GB, which leaves about 500MB free/available memory.

-

Note: If you’re wondering why such low numbers (e.g. < 15 number of requests per second), it is because the “calc.php” is a custom script that is very resource hungry – it connects to MySQL and selects over 9,000 records; retrieves the ID (an integer) and divides that number with “pi” – plus a few other mathematical functions.

The script itself requires a minimum of 128MB memory to run.

-
Here's a screenshot of the test result for Config #1.

```
Time taken for tests: 20.682 seconds
Complete requests: 300
Failed requests: 29
  (Connect: 0, Receive: 0, Length: 29, Exceptions: 0)
Write errors: 0
Total transferred: 90940468 bytes
HTML transferred: 90888868 bytes
Requests per second: 14.51 [#/sec] (mean)
Time per request: 344.693 [ms] (mean)
Time per request: 68.939 [ms] (mean, across all
Transfer rate: 4294.11 [Kbytes/sec] received

Connection Times (ms)
              min    mean[+/-sd] median    max
Connect:        0      0   0.1      0      1
Processing:    226    343   73.5    333    592
Waiting:       187    294   69.4    282    521
Total:         226    344   73.5    333    592
```

Let's see what happens when we increase the concurrency level from **5 to 25**, i.e. there are 25 simultaneous requests to this CPU intensive script (using config #1):

```
# ab -n300 -c25 http://donkeyvps.com/calc.php
```

The server ran out of memory and began to swap! (313MB out of 512MB swap space)

```
 1  [||||||||||||||||| 67.8%]   Tasks:
 2  [||||||||||||||||| 67.8%]   Load av
 3  [||||||||||||||||| 67.8%]   Uptime:
 4  [||||||||||||||||| 67.8%]
Mem[|||||||||||||||||1534/1536MB]
Swp[||||||||||||||||| 313/512MB]

USER      RES  CPU%  MEM%  Command
donkey    72568 20.2   4.6  php-fpm: pool donkey
donkey    72336  9.6   4.6  php-fpm: pool donkey
donkey    72176  8.2   4.6  php-fpm: pool donkey
donkey    72144  9.2   4.6  php-fpm: pool donkey
donkey    71884  8.7   4.6  php-fpm: pool donkey
```

The number of “requests/sec” dropped from 14.51 requests/sec to just 8.21 requests/sec. The time taken to complete the test increased by a whopping 17.21 seconds (from 20.68 seconds to 37.89 seconds) because the server had started to swap!

Test Case (B) - A typical Wordpress page with 3 images

This is a more “realistic” benchmark example - Wordpress page with some images

ab -n500 -c500 <http://donkeyvps.com/performance-bench/>

Note that this time, we're sending **500 requests** (with 500 concurrent users) to the WordPress page using the the same Config #1 and Config #2 as before. (In Test Case A - the concurrency of '25' was sufficient to cause the server to swap. But not so in this case).

Test Results

Configuration 1	Configuration 2
Memory usage: <200 MB	Memory usage: < 200MB
Time taken: 6.34 seconds	Time taken: 5.74 seconds.
<u>Num</u> of request/sec: 78.8	<u>Num</u> of request/sec: 87.18

Our server was able to serve between 79 – 87 requests per second while using less than 200MB of memory. Total time taken was 6.34 sec and 5.74 sec respectively.

In this scenario, the higher configuration of PHP-FPM (Config #2) gives better performance. This is because of Config #2 has more child processes (pm.max_children=100) to handle the higher concurrency of 500 (compared to the concurrency of “5” in scenario A).

A higher concurrency requires more processes/threads serve all the client requests.

AB bench Result (Config #1)

```
Concurrency Level:      500
Time taken for tests:    6.345288 seconds
Complete requests:      500
Failed requests:         0
Write errors:            0
Total transferred:      6643500 bytes
HTML transferred:       6512500 bytes
Requests per second:    78.80 [#/sec] (mean)
Time per request:       6345.288 [ms] (mean)
Time per request:       12.691 [ms] (mean, across all concurrent requests)
Transfer rate:          1022.33 [Kbytes/sec] received

Connection Times (ms)
              min      mean[+/-sd] median    max
Connect:        12      15   2.0      16      19
Processing:    266 3184 1654.3    3215    6312
Waiting:       265 3183 1654.2    3214    6310
Total:         285 3200 1652.7    3231    6326
```

AB bench Result (Config #2)


```

Concurrency Level:      500
Time taken for tests:    5.735479 seconds
Complete requests:      500
Failed requests:         4
    (Connect: 0, Length: 4, Exceptions: 0)
Write errors:           0
Non-2xx responses:      4
Total transferred:      6593232 bytes
HTML transferred:       6462548 bytes
Requests per second:    87.18 [#/sec] (mean)
Time per request:       5735.479 [ms] (mean)
Time per request:       11.471 [ms] (mean, across all connections)
Transfer rate:          1122.49 [Kbytes/sec]

Connection Times (ms)
              min  mean[+/-sd] median  max
Connect:        12   13   1.1    14    15
Processing:    106 2890 1691.0  2825  5711
Total:          118 2903 1692.1  2839  5726

```

Some observations/tips when tuning your server for performance:

From these 2 simple tests, PHP-FPM “Config #1” performed better in **Test Case A** where the concurrency is lower, but the script (calc.php) is CPU intensive. In contrast, it performed poorer in scenario 2 (Wordpress test) where the page is less resource intensive and the concurrency is much higher (500 concurrent requests versus 5) .

In your own environment, you need to perform your own tests because your applications, your traffic, etc will be different from others.

Some general Tuning tips

- Start with the default values. Tweak 1 or 2 parameter at a time.
- If your web-applications/websites is resource hungry (i.e. uses a lot of memory and CPU), then set the settings lower to prevent the server from swapping!

- If the reverse is true, e.g. your web-applications/websites are “normal” (e.g. wordpress sites), then a higher settings will probably yield better performance for high traffic/ concurrent requests.
- Make sure your server does NOT begin to swap because of insufficient memory. This is the #1 performance killer.

When running the “ab” tests, have 2 terminal windows open to view the Apache and PHP-FPM error-log files. It will give you clues as to what is happening.

Terminal #1: Apache Error log:

```
# tail -f /var/log/httpd/error_log
```

```
[Thu May 22 11:06:21 2014] [error] server reached
MaxClients setting, consider raising the MaxClients setting
```

Terminal #2: PHP-FPM Error log

```
# tail -f /var/log/php-fpm/error.log
```

```
[22-May-2014 07:32:45] WARNING: [pool donkey] seems busy (you may need to increase
pm.start_servers, or pm.min/max_spare_servers), spawning 32 children, there are 0
idle, and 15 total children
```

From these error logs, you’ll get an idea of what needs to be tweak and tuned.

Another useful log file to monitor is the /var/log/messages.

If the server ran out of memory, it will invoked the 'OOM' (Out-Of-Memory) killer process to terminate some processes and record the event in this log file. For example, if the settings are too high (in php-fpm pools or the httpd.conf file), it can cause the server to run out of memory as we have seen earlier.

```
# cat /var/log/messages | egrep -i 'oom'
```



```
May 22 22:14:06 srv10 kernel: [587292.612814] Out of memory
in UB 1193: OOM killed process 528 (lfd) score 0 vm:12216kB,
  rss:1892kB, swap:5872kB
May 22 22:14:06 srv10 kernel: [587292.671159] Out of memory
in UB 1193: OOM killed process 908 (nginx) score 0 vm:13384k
B, rss:4220kB, swap:1788kB
May 22 22:14:06 srv10 kernel: [587292.682639] Out of memory
in UB 1193: OOM killed process 1231 (httpd) score 0 vm:29141
6kB, rss:1880kB, swap:3056kB
May 22 22:14:06 srv10 kernel: [587292.711291] Out of memory
in UB 1193: OOM killed process 2063 (httpd) score 0 vm:29128
8kB, rss:2020kB, swap:2920kB
May 22 22:14:06 srv10 kernel: [587292.756403] Out of memory
in UB 1193: OOM killed process 2162 (httpd) score 0 vm:29128
8kB, rss:1908kB, swap:3012kB
```

If you see something like the above –“OOM killed process ...” that means your server ran out of memory. So, always start with smaller/conservative numbers and work your way up!

Chapter 8: Installing Nginx server as Reverse-Proxy

Nginx is a full featured, lightning fast Web-server that can serve as a replacement web-server to Apache, or it serve as a "reverse-proxy" (front-end) to Apache. It was designed from ground-up for speed and performance.

A quick note about why we are considering using Nginx as a Reverse-Proxy. In most hosting environment, the PHP SAPI used is "mod_suPHP" to provide the extra layer of security and isolation on the server. This means, every Apache child process loads the entire PHP interpreter and is therefore quite 'bloated'.

A client requesting a webpage that contains just plain html and images will still invoke the same 'bloated' Apache child-process which is in-efficient and slow.

This is where a "reverse-proxy" such as Nginx comes in. It will handle all the "static" content requests from clients and forwards only dynamic page (e.g contains php codes) requests to Apache. This speed things up considerably.

In our case, since we already have the 'separation' between Apache and PHP (via PHP-FPM), having Nginx as the front-end reverse proxy is optional.

In my own tests, it does provide a slight improvement in performance under normal to slightly heavy load, but it won't be 'noticeable' to the average web-visitor. So this chapter is provided as an "optional" bonus – should you want to setup Nginx as as reverse-proxy in front of Apache.

Ok, with that side note out of the way, let's dive in and install Nginx up as a reverse-proxy to Apache. We won't be going

into all the details of the configuration directives as there are just too many of them. There are many (thick) volumes written about Nginx - a short chapter like this will not do justice to all the capability of Nginx.

Installing Nginx is fairly straightforward - the good folks at Nginx have kindly setup the software repository and instructions at:

The first step is to download the appropriate rpm file for your CentOS version. In my case, I'm using CentOS 6, 32-bit

Step 1: Create the 'nginx repo'

Create the nginx.repo file in /etc/yum.repos.d/ and paste the following code into the file.

```
# vi nginx.repo

[nginx]
name=nginx repo
baseurl=http://nginx.org/packages/centos/6/$basearch/
gpgcheck=0
enabled=1
```

NOTE: If you're running RedHat's (RHEL) Enterprise Linux, then replace 'centos' with 'rhel'. If you're using CentOS 7 (or RHEL 7), then replace '6' with '7'. Once the nginx.repo has been created, you can install it via:

```
# yum install nginx
```

Step 2: Editing the Nginx configuration files for reverse-proxy

The nginx's configuration files are located at /etc/nginx/

```
# cd /etc/nginx && vi nginx.conf
```

Change: worker_processes 2;

The default value is 1. As a reverse-proxy to Apache, 2 worker processes is sufficient. But if you have a big multi-core server, you can set it to the same number cpu cores.

```
# cat /proc/cpuinfo | grep processor
```

Here's a sample listing of the 'nginx.conf' file:

```
#----- start nginx.conf -----#
user  nginx;
worker_processes  2;
error_log /var/log/nginx/error.log notice;
pid        /var/run/nginx.pid;
events {
    worker_connections 1024;
}
http {
    include      /etc/nginx/mime.types;
    default_type application/octet-stream;
    log_format main '$remote_addr - $remote_user [$time_local] "$request" '
                   '$status $body_bytes_sent "$http_referer" '
                   '"$http_user_agent" "$http_x_forwarded_for"';
    access_log /var/log/nginx/access.log main;
    sendfile      on;
    tcp_nopush     on;
    tcp_nodelay    on;
    types_hash_max_size 2048;
    server_tokens off;
    keepalive_timeout 3;
    gzip on;
    gzip_vary on;
    gzip_disable "MSIE [1-6]\.";
    gzip_proxied any;
    gzip_http_version 1.1;
    gzip_min_length 1000;
    gzip_comp_level 6;
    gzip_buffers 16 8k;
    gzip_types      text/plain text/xml text/css application/x-javascript
application/xml image/png image/x-icon image/gif image/jpeg application/javascript
application/xml+rss text/javascript application/atom+xml;
    include /etc/nginx/conf.d/*.conf;
}
#----- end -----#
```

Then, create a file called 'proxy_params' with the following content:

```
# --- Nginx Reverse Proxy_params ----- #
proxy_set_header Host $host;
proxy_set_header X-Real-IP $remote_addr;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

client_max_body_size 100M;
client_body_buffer_size 1m;
proxy_intercept_errors on;
proxy_buffering on;
proxy_buffer_size 128k;
proxy_buffers 256 16k;
proxy_busy_buffers_size 256k;
proxy_temp_file_write_size 256k;
```

```

proxy_max_temp_file_size 0;
proxy_read_timeout 300;
#----- end -----#

```

Save and exit. We will include this 'proxy_params' file from the Nginx virtual hosts configuration files. There should be an existing directory - /etc/nginx/conf.d/ - if not, please create it. Then, we will create our 2 fictitious virtual hosts (pony.conf and donkey.conf)

```
# vi /etc/nginx/conf.d/pony.conf
```

and paste the following codes into it:

```

# ----- start -----
server {
    listen    80; ## listen for ipv4; this line is default and implied
    server_name ponyvps.com www.ponyvps.com;
    root /home/pony/www;
    access_log /home/pony/logs/access_log main;
    index index.php index.html;

    location / {
        try_files $uri @proxy;
    }
    location @proxy {
        proxy_pass http://X.X.X.X:8080;
        include /etc/nginx/proxy_params;
    }
    location ~*.*\.php$ {
        proxy_pass http://X.X.X.X:8080;
        include /etc/nginx/proxy_params;
    }

    location ~*.*\.(3gp|gif|jpg|jpeg|png|ico|wmv|avi|asf|asx|mpg|mpeg|mp4|pls|mp3|mid|wav|swf|flv|html|htm|txt|js|css|exe|zip|tar|rar|gz|tgz|bz2|uha|7z|doc|docx|xls|xlsx|pdf|iso)$ {
        expires 1M;
        try_files $uri @proxy;
    }
    error_page 404 /404.html;
    error_page 500 502 503 504 /50x.html;
    location = /50x.html {
        root /usr/share/nginx/html;
    }
}
# ----- end -----

```

```
# vi /etc/nginx/vhosts/donkey.conf
```

```

# ----- start -----
server {
    listen    80 ;

```

```

server_name donkeyvps.com www.donkeyvps.com;
root /home/donkey/www;
index index.php index.html;
access_log /home/donkey/logs/access_log main;
location / {
    try_files $uri @proxy;
}
location @proxy {
    proxy_pass http://X.X.X.X:8080;
    include /etc/nginx/proxy_params;
}
location ~*\.php$ {
    proxy_pass http://X.X.X.X:8080;
    include /etc/nginx/proxy_params;
}
location ~*\.
(3gp|gif|jpg|jpeg|png|ico|wmv|avi|asf|asx|mpg|mpeg|mp4|pls|mp3|mid|wav|swf|flv|html|h
tm|txt|js|css|exe|zip|tar|rar|gz|tgz|bz2|uha|7z|doc|docx|xls|xlsx|pdf|iso)$ {
    expires 1M;
    try_files $uri @proxy;
}
error_page 404 /404.html;
error_page 500 502 503 504 /50x.html;
location = /50x.html {
    root /usr/share/nginx/html;
}
}
# ----- end -----

```

*NOTE: replace all 'X.X.X.X' with your server's IP address in the pony.conf and donkey.conf files above, as well as the /etc/httpd/conf/httpd.conf file.

Step 3: Edit Apache's httpd.conf to listen to port 8080

Edit the /etc/httpd/conf/httpd.conf file and change all occurrences of port 80 to 8080

```

Listen 8080
NameVirtualHost X.X.X.X:8080
<VirtualHost X.X.X.X:8080>

```

Step 4: Install mod_rpaf

When using Nginx's as the Front-End reverse-proxy, we need to configure/tell Apache to log the original requester's IP address, and not the IP address of the localhost (127.0.0.1).

To do this, we need to install the 'mod_rpaf' module. We can compile from source, or an easier way is to install it via "yum" from the "Atomic" repository. Go to:

<http://www6.atomicorp.com/channels/atomic/centos/6/>

The 'i386' directory is for the 32-bit architecture. The 'x86_64' is for the modern 64-bit.

An easier method is to use Atomic's automatic installer to install the repo:

```
# wget -q -O - http://www.atomicorp.com/installers/atomic | sh
```

You'll be prompted to agree with the Atomic repo license. Press the enter key to accept.

This will create the 'atomic.repo' file in the /etc/yum.repos.d/.

Edit this file:

```
# vi /etc/yum.repos.d/atomic.repo
```

Change the "enabled = 1" to "enabled = 0"

```
# Name: Atomic Rocket Turtle RPM Repository
# URL: http://www.atomicrocketturtle.com/
# Note: This isn't covered by ASL support
[atomic]
name = CentOS / Red Hat Enterprise Linux
mirrorlist = http://updates.atomicorp.com
enabled = 0
priority = 1
protect = 0
```

Reason being we don't want the Atomic repo to be active in case we accidentally over-write our other system software when we issue the "yum update" command.

We only want the Atomic repo for the "mod_rpaf" module.

Once we've disabled this repo ("enabled = 0"), we can proceed to install mod_rpaf by manually enabling this repo during the yum install command.

```
# yum --enablerepo=atomic install mod_rpaf
```

Step 5: Restart Apache and Nginx

Final step – restart Apache (which should be listening on port 8080 now) and start Nginx

```
# service httpd restart && service nginx start
```

Checking the Apache access log file:

```
127.0.0.1 - - [05/Nov/2014:09:58:10 -0500] "GET /manual/ko/sitemap.html
127.0.0.1 - - [05/Nov/2014:09:58:10 -0500] "GET /404.html HTTP/1.0" 404
-----
123.136.107.69 - - [05/Nov/2014:10:00:26 -0500] "GET / HTTP/1.0" 304 -
```

Prior to installing the 'mod_rpaf', the log file shows "127.0.0.1" (which is the IP addr of the Nginx server. After installing "mod_rpaf", Apache is able to correctly log the actual IP address of the web-visitor.

Troubleshooting Tips:

If you had change the directory permissions of /home/pony & /home/donkey to "750" to prevent PHP scripts from accessing another users' file, you will also have to add the userID "nginx" as a user to the secondary group "pony" and "donkey", e.g.

```
# usermod -a -G pony nginx
# usermod -a -G donkey nginx
```

This will enable the Nginx server to access static files, e.g. html, javascripts, css and image files in those users' directory.

Another thing to take note of is that some tutorials advocate using the IP (127.0.0.1) in the 'proxy_pass' directive in Nginx, e.g.

```
proxy_pass http://127.0.0.1:8080;
```

You can do that, but if you do, then you will need to change all the <VirtualHost> container in the 'httpd.conf' file to

127.0.0.1 as well (instead of the server's IP address). E.g.

```
<VirtualHost 127.0.0.1:8080> and  
Listen 127.0.0.1:8080
```

Alternatively, if you want to be 'lazy' - you can use the wildcard, * in Apache config file, e.g.

```
<VirtualHost *:8080> and  
Listen *:8080
```

Closing thoughts on Nginx

Some folks advocate ditching the Apache webserver completely and use the Nginx as the primary web-server. Although Nginx was designed from ground up for speed, there is a rather steep learning curve compared to Apache which has been around for a much longer time.

Also, nearly all web-applications will “work-out-of-the-box” with Apache. If you are hard-pressed for time and resources, using Nginx as a reverse-proxy is the way to go to boost server performance.

But if you have the time and resources, then by all means, explore this option, but take note of some of the factors listed below before making the switch to replace Apache with Nginx.

If your server currently has many websites and you use the .htaccess files extensively, then it is not advisable. The 'labor' cost of switching over to Nginx and converting all the configuration to the new format (as well as testing them to ensure nothing breaks) will be enormous.

If you are providing hosting services to your clients with Cpanel (or some other commercial control panel), then it is also not advisable, unless you want to spend all your time answering support tickets and emails from irrate customers.

If your server is for your own use and you are serving mainly static files (html, css, javascript, images, etc), then yes, do consider switching over to Nginx.

If Server resources (memory, CPU) are scarce, then maybe - Nginx tends to have a smaller footprint compared to Apache - although, in my opinion, if you're familiar with Apache, you could try compiling Apache from source with minimal static modules. You can make Apache as 'lean and mean' OR as 'bloated' as you want.

If you are thinking of using Nginx as your PHP server optimization, then I'm afraid you're 'barking up the wrong tree'.

The ideal scenario is to separate the 'application-server' (i.e. PHP-FPM) from the 'web-server' (Apache or Nginx). You already have the knowledge and skills to setup PHP-FPM from this book.

Summary: Book Errata & Support

Thank you for reading this book and I hope this book has given you a solid foundation in understanding the various PHP SAPIs, Apache Handlers and more importantly, the practical steps to setup PHP-FPM and Nginx reverse proxy on your server.

The chapters on Security (***Securing Apache*** and ***Securing PHP***) should be implemented, especially if you are providing hosting services to the public. If you are the only user in the system, then it is not so vital, but nevertheless, still a good practice to put in place. One can never be too lax with security.

But in any case, I've created a dedicated to page here for Book Erratas:

<http://ilovevirtualmin.com/book-3>

If you've found this book helpful, I'd appreciate it very much if you could leave a review or comment here so that others may also benefit from it.

<http://ilovevirtualmin.com/go/book3.php>

If you've any questions or if there are other topics which you'd like to see in the future, do let me know.



Email: adrian.ling@gmail.com

My Other Kindle Books

All my Kindle books in the Practical System Admin Guide Series can be found here:

<http://www.amazon.com/-/e/B00WFQH438>

Webmin & Virtualmin: A quick and Practical Guide to setting up and configuring this powerful Web-Based Server Control panel.

While this Open Source Control Panel is not as slick as the commercial Cpanel, it is just as powerful and more flexible in helping you manage your servers from the web-browser

Virtualmin Tips and Hacks: How to Setup PHP-FPM Automatically and Run Nginx Reverse Proxy. This book covers the more advance topics and shows you how to configure Virtualmin to automatically setup new virtual hosts with PHP-FPM.

Special Bonus worth \$49! It comes with custom shell scripts to help you automate the process

Appendix A: An Overview of Apache MPM

The 'core' engine in Apache is called the 'MPM' - "Multi Processing Modules". Although it's called 'Modules', the MPM is actually static* and is compiled into the Apache binary itself. The MPM is the Apache code that handles all the incoming requests (e.g. the complete HTTP session from listening on the network, responding to client requests and how to handle the Apache child-processes and threads).

Different MPMs will handle the requests differently. You cannot choose to change the MPM once you've installed the Apache webserver without re-compiling or re-installing it again. (**note:** MPM being 'static' is true for Apache 2.2 and earlier. In 2.4, the MPM can be loaded as a 'module' during run-time)

On Linux/Unix based machines, Apache has 3 main MPMs to choose from; Prefork, Worker, and Event. (There are other 3rd-party MPMs such as "itk" but we will just stick to the official MPMs in this book)

Prefork MPM

The prefork MPM uses multiple child processes, each child-process has 1 thread only. Each child process handles one connection at a time. This is the original Apache MPM and is generally more stable since it does not use threads. This makes it ideal where you need to run 3rd party modules which are non-thread-safe. This MPM uses more memory compared to the 'worker' MPM

In most hosting environment, this is the preferred MPM because it is thread-safe and provides the most stable hosting environment (at the slight expense of increased memory usage)

<http://httpd.apache.org/docs/2.2/mod/prefork.html>

Worker MPM

A single control process (the parent) is responsible for launching child processes. Each child process creates a fixed number of server threads as specified in the ThreadsPerChild directive, as well as a listener thread which listens for connections and passes them to a server thread for processing when they arrive.

Apache always tries to maintain a pool of spare or idle server threads, which stand ready to serve incoming requests. In this way, clients do not need to wait for a new threads or processes to be created before their requests can be served. The number of processes that will initially launch is set by the StartServers directive. During operation, Apache assesses the total number of idle threads in all processes, and forks or kills processes to keep this number within the boundaries specified by MinSpareThreads and MaxSpareThreads.

This MPM is generally better for high-traffic servers because of its smaller memory footprint. But be aware of potential problems if you are using 3rd party Apache modules that may not be thread-safe.

For more info:
<http://httpd.apache.org/docs/2.2/mod/worker.html>

Event MPM

This is the latest MPM from Apache which has been in 'beta' status for a long time. It has been finally declared stable in Apache 2.4. The way it function is similar to the Worker MPM, but with one main difference - it will assign a thread to

a request, but NOT the entire http connection (as in the case of the Worker MPM).

This is useful in cases where the KeepAlive directive is enabled and the KeepAliveTimeout is set to a long time, eg. 300seconds. This will tie-up the entire thread (in the Worker MPM) even if the connection is idle.

But in the case of the Event MPM, the thread is used to respond to requests only and frees up immediately after that request has been fulfilled, regardless of the actual HTTP connection (that is still being kept alive). This means fewer threads (within each Child processes) is required and therefore, better memory usage. Think of 'Event' MPM as an enhanced version of the Worker MPM.

<http://httpd.apache.org/docs/2.2/mod/event.html>

Here's a quick side-by-side comparison between the 2 popular MPMs (in Apache 2.2)

MPM: <u>prefork</u>	MPM: worker
Multi-processes; 1 thread per process	Multi-processes; multiple threads per process
Each request/connection is handled by a process	Each request/connection is handled by a thread
This MPM is chosen mainly for stability and security.	This MPM is chosen for better performance and lower memory consumption.

<p>Summary:</p> <p>This MPM is used on sites that need compatibility with non-thread-safe libraries.</p> <p>Because each request is handled by a different process, it provides the best isolation – so that a problem with a single request will not affect any other.</p> <p>This MPM is very self-regulating, so it is rarely necessary to adjust its configuration directives.</p> <p>Most important is that <u>MaxClients</u> – it should be big enough to handle as many simultaneous requests as you expect to receive, but small enough to assure that there is enough physical RAM for all processes.</p>	<p>Summary:</p> <p>This is a hybrid multi-process, multi-threaded server and uses threads to serve requests.</p> <p>Because of this, it is able to serve a large number of requests with fewer system resources compared to 'prefork' MPM.</p> <p>Note: it retains much of the stability of the 'prefork' MPM by keeping multiple processes available, each with many threads.</p> <p>The important directives used to control this MPM are <u>ThreadsPerChild</u>, which controls the number of threads deployed by each child process and <u>MaxClients</u>, which controls the maximum total number of threads that may be launched.</p>
---	---

For more info about the MPM:
<http://httpd.apache.org/docs/2.2/mpm.html>

-----sidebar-----

Processes vs Threads

- * Processes:
 - own copy of data structures
 - shares program code, shared memory
 - context switches are expensive
- * Thread:
 - runs within a process
 - shares process environment
 - no context switch

Appendix B: Apache 2.4 vs Apache 2.2

Apache 2.4.1 was released on Feb 2012. The current/latest version of Apache 2.4 (as of this writing) was released in January 2015 (version 2.4.12)

The main improvements (in my opinion) over Apache 2.2 are its ability to:

Choose the MPM at run-time as loadable modules. If you recall, in Apache 2.2 and earlier, you have to choose a specific MPM (either 'prefork' or 'worker') when you compile and build Apache. That MPM is built into the Apache engine itself and does not change unless you re-compile and re-build Apache again. In version 2.4, multiple MPM can now be built as loadable modules at compile time, and you can choose which MPM to load in the configuration file.

The “Event” MPM has matured and is now considered stable and fully supported.

Ability to specify the ‘KeepAlive’ timeout in milliseconds instead of seconds.

Find-grained controls on the “Override” configuration directives in the .htaccess file.

Better memory usage and control compared to Apache 2.2

Apache 2.4 also introduces a host of new modules (e.g. mod_proxy_fcgi - for handling FastCGI protocol) and utilities such as the 'fcgistarter' (to start a FastCGI program).

For a complete list of enhancements, please visit:

http://httpd.apache.org/docs/current/new_features_2_4.html

If you plan to upgrade your current Apache 2.2 to Apache 2.4, please be aware of the following important run-time

changes which you need to update in the apache configuration file (httpd.conf).

One of the important changes is the authorization configuration, for example:

- In Apache 2.2, access control based on client hostname or IP address was done using the 'Order', 'Allow' and 'Deny' directives.
- In Apache 2.4, access control is done using the new module 'mod_authz_host' although compatibility with the old configuration is provided in the 'mod_access_compat' module.

<http://httpd.apache.org/docs/2.4/upgrading.html>

Appendix C: Apache's Directives

The Apache directives are placed in the main configuration file: '/etc/httpd/conf/httpd.conf'

From the main configuration file, there are various 'include' statements to include other configuration files which 'groups' similar directives.

Apache has over 400+ directives which are rather daunting to the beginner – but you've already know the basics, so if you want to dig in deeper, please visit the official Apache's documentation site:

For Apache 2.2.x

- <http://httpd.apache.org/docs/2.2/mod/directives.html>

For Apache 2.4.x

- <http://httpd.apache.org/docs/2.4/mod/directives.html>

Keep in mind, that generally, the configuration directives can be categorized into:

A) The Global Environments (GE)

The directives in this section controls the overall operation of the Apache server.

B) The Server Config (SC)

The directives in this section relates to the main server and provides the default values for virtual host containers.

C) The Virtual Host Configuration. (VH)

The virtual host directives are used to run multiple websites (different domains) on the same physical server, sharing the same IP address.

Let's look at some of the essential directives in these categories now:

A) The Global Environment (GE)

ServerTokens ProductOnly

- This directive tells Apache what to send in the header field back to the client. The information can be generic, e.g. "Apache" or a detailed description such as "Apache/2.2.4 (CentOS)" *Suggested setting is: "ProductOnly" which displays only report 'Apache'. This is for security reason*

ServerName mysite.com

- This is the hostname and port that the server uses to identify itself. If this directive is not set, then the server will try to deduce the hostname by performing a reverse IP lookup which takes time.
- *So always set a 'ServerName'. This directive is mandatory in the 'VirtualHost' container.*

ServerAdmin admin@mysite.com

- *The email address of the admin in charge of this server*

ServerRoot /etc/httpd

- This is the base directory where Apache was installed. Default location is at /usr/local/apache if you install from source. For CentOS (if installed via the 'yum' command, it will default to '/etc/httpd' with symlinks to /usr/lib/httpd/modules/ and /var/log/httpd/ for the 'modules' and 'logs' subdirectories.
- *Relative paths in other configuration directives in the 'httpd.conf' (e.g. "LoadModule" or "Include") are taken as relative to this ServerRoot directory*

DocumentRoot /home/userID

- *This directive tells Apache where to find and serve files (webpages, php scripts, etc) to the users that requests for them. Note: the DocumentRoot must be specified WITHOUT a trailing slash '/'*

Listen 80

- *Tells Apache which port to listen to. By default, all web servers, including Apache listens to port 80. If you are using Nginx as front-end reverse-proxy, we need to change the "Listen" directive to another port, e.g. 8080*

User apache

- *When the Apache server starts, the main (parent) process is owned by root and runs with root privileges. This parent process will then spawn 'child-server' processes with the '**user**' specified in this directive. Defaults are usually the user 'apache', 'httpd' or 'nobody'*

Group apache

- This is the group that the Apache child-server belongs to. *Typically, it's the same as the 'user'*

KeepAlive On / Off

- This directive is a double-edged sword. It tells the Apache to keep a HTTP session 'alive' so that a client (browser) can send multiple requests over the same TCP connection. In some cases, setting KeepAlive 'On' has been shown to improve speed by almost 50% for html documents with multiple images.
- Note: setting up new TCP connections and closing existing TCP connections are expensive, that is why in some cases, telling Apache to keep existing TCP connections 'alive' will speed things up. If this is set to "On", then make sure this value is low - e.g. 3-5 seconds to prevent idle clients from tying up server resources.
- *Tip: If you are using Nginx as reverse-proxy, then turn this off and turn 'KeepAlive On' in Nginx instead*

Timeout 60

- This is the length of time Apache will wait for I/O in various circumstances such as:
 - When reading data from the client, the length of time to wait for a TCP packet to arrive if the read buffer is empty.
 - When writing data to the client, the length of time to wait for an

acknowledgement of a packet if the send buffer is full.

- In `mod_cgi`, the length of time to wait for output from a CGI script.
- *TIP: Do not set this value too high as nobody will wait more than 60 seconds for a page to load.*

KeepAliveTimeout 3

- *This is the number of seconds Apache will wait for a subsequent request before closing a persistent connection. Once a request has been received, the KeepAliveTimeout value applies. If the client does not issue a subsequent request within this time frame, Apache will close the connection. Keep this value low (3-5 secs) to prevent an idle client from tying up server resources*

MaxKeepAliveRequests 500

- *This directive limits the number of requests allowed PER connection that is kept alive. If 'KeepAlive' is ON, set this value high to fully make use of the TCP connection. Any value above 200 is ok*

Suexec Off | On

- *If set to 'On' tells Apache to run a script (e.g. php page) as the 'owner' of that page. It is 'On' by default if the suexec binary exists with proper owner and mode. If the suexec binary doesn't exist and this directive is set to 'On', Apache will fail to start.*
- *This directive is only applicable (and useful) if you opted to use the 'mod_suPHP' handler. In the case*

PHP-FPM, it does not apply.

B) The Server Config (SC)

This set of “Server Config” directives controls how many child-server processes are spawned and managed to handle the server load. In CentOS, when you install Apache via the ‘yum’ utility, it installs all the 3 MPMs (prefork, worker and event). The default “active” MPM is the ‘prefork’ MPM.

We can easily switch between the different MPMs that we want to use by deleting the ‘httpd’ symlink and re-linked it to the appropriate MPM binary.

Ok, let’s look at the server directives. If you are using the “prefork” MPM, then the directives enclosed within the <IfModule prefork.c> container will be used. If using the “worker” MPM, then the directives within the <IfModule worker.c> container will be used.

Let's go through some of the commonly used directives and their description below:

StartServers 2

- *This directive sets the number of child-server processes that should be created on startup. In the “prefork” MPM, the number of StartServers you specify here will be (usually) the number of “httpd” processes that you will see with the “ps aux | grep httpd” command.*
- *In contrast, with the “worker” MPM, the num of StartServers we specify here is the initial number of child-server processes that the root process will spawn at startup. But during operation, Apache*

*assesses the total number of idle threads in all processes, and forks or kills processes to keep this number within the boundaries specified by **MinSpareThreads** and **MaxSpareThreads**. This means the “ps aux|grep httpd” may (usually) not show 3 httpd processes*

MaxClients 250

- *This directive sets the maximum number of concurrent connections that Apache will service simultaneously. Any connection attempts over this limit will be queued, up to the number specified in the 'ListenBacklog' directive.*
- *In the 'prefork' MPM, the MaxClients translate into the max number of child-server processes that will be created to serve client requests. The default value is 256. If you need to increase it, you will also need to increase the ServerLimit. (TIP: set the “ServerLimit” to the same value as MaxClients to prevent accidental error)*
- *In the 'worker' PMP, MaxClients limit the number of threads that are available to serve client requests. The default is 16 (ServerLimit) multiplied by 25 (ThreadsPerChild).*

ServerLimit 250

- *Typically, you don't need to edit this value unless you need to go above the default max value (256) in the 'MaxClients' (prefork MPM). For example, if you*

set MaxClients 300, then you need to set ServerLimit 300 (or higher) as well.

- *Take note that if you set this directive to a value much higher than necessary, the extra unused memory will still be allocated. This may cause the system (server) to become unstable. Apache may also refuse to start if the ServerLimit and MaxClients values are set too high*

MinSpareServers 5

- *This directive tells Apache to keep at least this number of idle child-server processes. An idle process is one that is not handling any client request. If there is fewer than this number, the Apache parent process will spawn new child-server processes.*
- *This is an important because it ensures that if there is a sudden spike in website traffic, there are sufficient child-server processes to handle the client requests.*
- *BUT, setting this value too high may cause Apache to use up too much memory can cause the physical server to run out of memory and start swapping. That is a huge performance hit!*

MaxSpareServers 5 (this is for the “prefork” MPM only)

- *This directive limits the maximum number of idle child-server processes that are running. An idle process is one that is not handling any client*

request. If there are more than the specified MaxSpareServers, then Apache will terminate the extra child-server processes

ThreadsPerChild 25 (this is for the “Worker” MPM only)

- *This directive instructs Apache how many threads each child-server process should create. Each child-server process creates this number of threads during startup and does not create any more. If there are multiple child-processes running in a 'worker' MPM, the total number of threads should be high enough to handle average load on the server. The default value is 25*

MinSpareThreads 5 (“Worker” MPM)

- *This directive Apache to keep this total number of spare (idle) threads on a server-wide basis. If there aren't enough idle threads in the server, then Apache will spawn new child-server processes until the number of idle threads is greater than this number*

MaxSpareThreads 20 (“Worker” MPM)

- *This is the max. number of idle threads on server-wide basis. If there are too many idle threads in the server, then Apache will terminate child-processes until the number of idle threads falls below this value*

MaxRequestsPerChild 2000

- *This directive sets max number of requests a child-server process will handle before terminating. The default is zero, which means it will never die.*
- *It is not recommend to set this to zero as there are many poorly coded applications (e.g. various Plugins for Wordpress, Joomla, Drupal, etc) that can cause memory leak.*
- *Set it to a high value, e.g. 2000 or more, depending on how much traffic your server receives*

C) The Virtual Host Configuration. (VH)

The Apache's directives may be applied to the entire server or they may be restricted to apply only to selected directories, files, hosts and even URLs. Apache has the following section containers (one of which is the <VirtualHost>)

- <Directory>
- <DirectoryMatch>
- <Files>
- <FilesMatch>
- <IfModule>
- <IfDefine>
- <IfVersion>
- <Location>
- <LocationMatch>
- <Proxy>
- <ProxyMatch>
- <VirtualHost>

Think of a container as a 'section' or a group of directives that will be executed only if it matches the condition specified by the container, for example:

```
<IfModule prefork.c>
... if the prefork MPM is loaded, then run the directives here
</IfModule>

<IfModule worker.c>
... if the worker MPM is loaded, then run the directives here
</IfModule>
```

These containers can be nested as well. For example, here's a basic `<VirtualHost>` container with nested `<Directory>` container within it.

```
<VirtualHost 192.168.5.10:80>
  ServerAdmin root@srv8.mydomain.com
  DocumentRoot /home/user8/public_html
  ServerName mydomain.com
  ErrorLog /home/user8/logs/error_log
  CustomLog /home/user8/access_log common
  <Directory "/home/user8/public_html">
    Options Indexes FollowSymLinks SymLinksIfOwnerMatch
    AllowOverride All
  </Directory>
</VirtualHost>
```

The most commonly used section containers you will see in the `httpd.conf` file are:

`<IfModule>`

- If this module is available, then run the directives enclosed within this container

`<VirtualHost>`

- It encloses directives that apply to this particular virtual host

`<Directory>`

- Directives within this section container apply the named directory and subdirectories. This is the physical path of the filesystem on the server, e.g. `<Directory /var/www/html >`

`<Files>`

- Directives enclosed in this container apply only to the specified filename, regardless of the file's location on the filesystem, e.g. `<Files secret.txt>`

For more details on all these containers:

<http://httpd.apache.org/docs/2.2/sections.html>

<http://httpd.apache.org/docs/2.4/sections.html>

Let's look at the 2 important directives that are often used in the `<Directory>` container:

1. The “Options” Directive

```
<Directory /var/www/html >
Options option1 option2 option3 ...
</Directory>
```

2. The “AllowOverride” directive

```
<Directory /var/www/html >
AllowOverride option1 option2 option3 ...
</Directory>
```

(1) The “Options” Directive has the following 'options'

Options	Description
All	This is the default. All options are enabled except for 'MultiViews'
None	Nothing is enabled within this directory and subdirectories
<u>MultiViews</u>	If mod_negotiation is enabled, this option will enable the 'negotiated content' to be displayed on the browser
<u>ExecCGI</u>	Enables the execution of CGI scripts via the "mod_cgi"
<u>FollowSymLinks</u>	Allows Apache to follow symbolic links in this directory.
<u>SymLinksIfOwnerMatch</u>	Apache will only follow symbolic (<u>sym</u>) links if the target file or directory is owned by the same user ID as the link. If you are providing hosting services to the clients, you should enable this for better account isolation and security for your hosting clients.
Includes	Allows server-side includes provided by the 'mod_include' module.
<u>IncludesNOEXEC</u>	Server-side includes are permitted except the #exec cmd and the # exec cgi which are disabled.
Indexes	If an url maps to a directory and no index file is found, then Apache will display a listing of all the files in that directory if 'mod_autoindex' is loaded. Otherwise, an error will occur.

Note: One can use the + symbol or the - symbol to add/remove options from directories. For example:

```
<Directory /var/www/html>
Options Indexes ExecCGI
</Directory>
```

```
<Directory /var/www/html>
Options -Indexes +SymLinksIfOwnerMatch
</Directory>
```

The directory /var/www/html will have the 'ExecCGI' and 'SymLinksIfOwnerMatch' options applied to it. The 'Indexes' option has been negated by the minus (-) symbol.

(2) The "AllowOverride" Directive has the following 'options'

Options	Description
All	Any directive which has the <u>.htaccess</u> Context is allowed in <u>.htaccess</u> files.
None	If set to None, the <u>.htaccess</u> file is completely ignored. Apache will not even attempt to read <u>.htaccess</u> files in the <u>filesystem</u> .
<u>AuthConfig</u>	Allow use of the authorization directives (<u>AuthDBMGroupFile</u> , <u>AuthDBMUserFile</u> , <u>AuthGroupFile</u> , <u>AuthName</u> , <u>AuthType</u> , etc)
<u>FileInfo</u>	Allow use of the directives controlling document types – this is important because it is required by 'mod_rewrite' (needed by <u>Wordpress</u> for the <u>SEO permalink URL format</u>) and 'mod_actions' (required for PHP-FPM)
Indexes	Allow use of the directives controlling directory indexing such as <u>AddDescription</u> , <u>AddIcon</u> , <u>AddIconByEncoding</u> , <u>IndexIgnore</u> , <u>AddIconByType</u> , <u>DefaultIcon</u> , <u>DirectoryIndex</u> , <u>HeaderName</u> , <u>IndexOptions</u> , <u>ReadmeName</u> , etc.
Limit	Allows the use of the directives controlling host access, i.e. <ul style="list-style-type: none"> • Allow • Deny • Order

For performance and security, set AllowOverride to None in the root <Directory /> block. Then, selectively create/nest <Directory> within the root "/" directory to place additional directives such as the .htaccess file. For example:

```
<Directory />
    Options SymLinksIfOwnerMatch
    AllowOverride None
</Directory>

<Directory "/var/www/html">
    Options +Indexes
    AllowOverride FileInfo
    Order allow,deny
    Allow from all
</Directory>
```

These directives are just the tip of the ice-berg – there are tons of different directive and configuration options, but knowing just these will put you ahead of many server Apache Administrators who merely rely on 'GUI' (web-based)

panels to manage their servers. You're one of the rare breed that goes beyond the GUI to master the command line!

For more detailed reading:

<http://httpd.apache.org/docs/>

<http://nginx.org/en/docs/>

Congratulations on making it to the end of this book. You should have a solid foundation to build your newfound knowledge and skills in Apache, PHP-FPM and Nginx. If you've found this book helpful, I'd appreciate it very much if you could leave a review or comment here so that others may also benefit from it.

<http://ilovevirtualmin.com/go/book3.php>

If you've any questions or if there are other topics which you'd like to see in the future, do let me know: Email:

adrian.ling@gmail.com



My Other Kindle Books

Check out my other Kindle books in the Practical System Admin Guide Series here:

<http://www.amazon.com/-/e/B00WFQH438>

Table of Contents

[Acknowledgements](#)

[Introduction](#)

[Chapter 1. The 'LAMP' stack](#)

[Chapter 2. Apache Handlers & PHP SAPI](#)

[Chapter 3. Instal](#)

[Chapter 4. Boost Performance with Opcode Cache](#)

[Chapter 5. Securing Apache](#)

[Chapter 6. Securing PHP \(php.ini\).](#)

[Chapter 7. Performance Tuning](#)

[Chapter 8: Installing Nginx server as Reverse-Proxy](#)

[Summary: Book Errata & Support](#)

[My Other Kindle Books](#)

[Appendix A: An Overview of Apache MPM](#)

[Appendix B: Apache 2.4 vs Apache 2.2](#)

[Appendix C: Apache's Directives](#)

[Unnamed](#)